Apuntes de Algoritmos y Estructuras de Datos
Programación III
Facultad de Informática - UNLP

Alejandro Santos
ale@ralo.com.ar

Curso 2013
(actualizado 29 de julio de 2013)

Índice general

Introducción

El presente apunte es una recopilación de explicaciones y exámenes parciales dados como parte de los cursos 2012-2013 de Programación III de Ingeniería en Computación, UNLP[1]. Actualmente se encuentra en estado de borrador, y puede contener $\epsilon \mathrm{r} r_o$ r^e`s` u omisiones, por lo que se recomienda consultar a los docentes ante cualquier duda o ambiguedad.

Puede contactarse con el autor por e-mail a: `ale@ralo.com.ar`.

[1] `http://www.info.unlp.edu.ar/`

Capítulo 1

Tiempo de Ejecución

1.1. Tiempo de Ejecución

Nos interesa analizar y comparar diferentes algoritmos para saber cuál es más eficiente.

Una forma de hacerlo es escribir el código de cada algoritmo, ejecutarlos con diferentes entradas y medir el tiempo de reloj que demora cada uno.

Por ejemplo, para ordenar una secuencia de números existen diferentes algoritmos: inserción, mergesort, etc. Se puede tomar cada uno de estos algoritmos y ejecutarlos con una secuencia de números, midiendo el tiempo que tarda cada uno.

Sin embargo, existe otra forma de comparar algoritmos, que se conoce como análisis asintótico. Consiste en analizar el código y convertirlo en una expresión matemática que nos diga el tiempo que demora cada uno en función de la cantidad de elementos que se está procesando.

De esta forma, ya no es necesario ejecutarlos, y nos permite comparar algoritmos en forma independiente de una plataforma en particular. En el caso de los algoritmos de ordenación, la función nos dará un valor en base a la cantidad de elementos que se están ordenando.

1.2. Análisis Asintótico BigOh

$f(n)$ es de $\mathcal{O}(g(n))$ si y solo si $\exists c$ tal que $f(n) \leq cg(n), \forall n \geq n_0$.

La definición de BigOh dice que una función $f(n)$ es de orden $g(n)$ si existe una constante c tal que la función $g(n)$ multiplicada por dicha constante acota por arriba a la función $f(n)$.

Cuando decimos acota queremos decir que todos los valores para los que la función está definida son mayores. Esto es, la función $g(n)$ acota a $f(n)$ si cada valor de $g(n)$ es mayor o igual que $f(n)$. Por ejemplo, la función $g(n) = n + 1$ acota por arriba a la función $f(n) = n$ ya que para todo $n \geq 0, f(n) \leq g(n)$.

Nuestro objetivo es encontrar un c que multiplicado por $g(n)$ haga que se cumpla la definición. Si ese c existe, $f(n)$ será de orden $g(n)$. Caso contrario, si no es posible encontrar c, $f(n)$ no es de orden $g(n)$.

El problema entonces se reduce a encontrar el c, que algunas veces es trivial y directo, y otras no lo es. Además, el hecho de no poder encontrarlo no quiere decir que no exista.

1.2.1. Por definición

Usando la definición de BigOh, una forma de descubrir si una función $f(n)$ es de orden $g(n)$ se traduce en realizar operaciones para despejar el valor de c, y descubrirlo en caso que exista, o llegar a un absurdo en caso que no exista.

1.2.2. Ejemplo 1, por definición Big-Oh

Ejemplo: ¿3^n es de $\mathcal{O}(2^n)$? Primero escribimos la definición de BigOh:

$$\exists c, f(n) \leq cg(n), \forall n \geq n_0 \tag{1.2.1}$$

Luego identificamos y reemplazamos las partes:

$$f(n) = 3^n \tag{1.2.2}$$
$$g(n) = 2^n \tag{1.2.3}$$

Una vez identificadas las partes podemos expresar la pregunta en forma de BigOh, donde tenemos que encontrar un c que cumpla lo siguiente:

$$3^n \leq c2^n, \forall n \geq n_0 \tag{1.2.4}$$

La segunda parte de la definición, que dice para todo n mayor o igual a n_0, es tan importante como la primera, y no hay que olvidarse de escribirlo. El valor de n_0 puede ser cualquier que nos ayude a hacer verdadera la desigualdad. Por ejemplo, en este caso se puede tomar $n_0 = 1$.

n_0 no puede ser negativo. Elegir un n_0 negativo significa que estamos calculando el tiempo de ejecución con una cantidad negativa de elementos, algo que no tiene sentido.

Eso también significa que las funciones siempre deben dar un valor positivo, dado que el valor de la función de la que se quiere calcular el orden representa tiempo de ejecución de un algoritmo. El tiempo debe ser siempre positivo, y por ejemplo decir que una función demora -3 segundos en procesar 10 elementos tampoco tiene sentido.

Se asume que 3^n es de $\mathcal{O}(2^n)$, y se pasa el 2^n dividiendo a la izquierda:

$$\frac{3^n}{2^n} \leq c, \forall n \geq n_0 \tag{1.2.5}$$
$$\left(\frac{3}{2}\right)^n \leq c, \forall n \geq n_0 \tag{1.2.6}$$

Se llega a un absurdo, ya que no es posible encontrar un número c para que se cumpla la desigualdad para todo n mayor que algún n_0 fijo. Por lo tanto es falso, 3^n no es de $\mathcal{O}(2^n)$.

Un análisis un poco más cercano nos muestra que la funcion $(\frac{3}{2})^n$ es siempre creciente hasta el infinito, donde el límite:

$$\lim_{n\to\infty} \left(\frac{3}{2}\right)^n \tag{1.2.7}$$

toma el valor infinito. Por ejemplo: para cualquier constante c fija, para algunos valores de n se cumple, pero para otros no. Con $c = 100$,

$$\left(\frac{3}{2}\right)^n \leq 100, \forall n \geq n_0 \tag{1.2.8}$$

Despejando el valor de n,

$$\log_{\frac{3}{2}} \left(\frac{3}{2}\right)^n \leq \log_{\frac{3}{2}} 100, \forall n \geq n_0 \tag{1.2.9}$$

$$\log_{\frac{3}{2}} \left(\frac{3}{2}\right)^n = n; \log_{\frac{3}{2}} 100 \approx 11{,}35 \tag{1.2.10}$$

$$n \leq 11{,}35, \forall n \geq n_0 \tag{1.2.11}$$

No es posible que n sea menor a 11,35 y a la vez n sea mayor a algún n_0, ya que n debe ser mayor que n_0 siempre.

1.2.3. Por regla de los polinomios

En las teorías se vieron algunas reglas para simplificar el trabajo:

Dado un polinomio $P(n)$ de grado k, sabemos que el orden del polinomio es $\mathcal{O}(n^k)$. Ejemplo: $P(n) = 2n^2 + 3n + 5$. Como el grado del polinomio es 2, el orden de $P(n)$ es $\mathcal{O}(n^2)$.

Esta regla solo se aplica a polinomios. Cualquier función que no lo sea habrá que desarrollarla por algún otro método. Ejemplo: $T(n) = n^{\frac{1}{2}} = \sqrt{n}$ no es un polinomio, y no puede aplicarse esta regla.

1.2.4. Por regla de la suma

Si $T_1(n) = \mathcal{O}(f(n))$ y $T_2(n) = \mathcal{O}(g(n))$ entonces:

$$T_1(n) + T_2(n) = max(\mathcal{O}(f(n)), \mathcal{O}(g(n))) \tag{1.2.12}$$

1.2.5. Otras reglas

$T(n) = \log^k n \to$ es $\mathcal{O}(n)$.

$T(n) = cte$, donde cte es una expresión constante que no depende del valor de $n \to$ es $\mathcal{O}(1)$.

$T(n) = cte \times f(n) \to$ es $\mathcal{O}(f(n))$.

1.2.6. Ejemplo 2, por definición Big-Oh

Calcular y demostrar el orden de $T(n)$ por definición:

$$T(n) = 2n^3 + 3n^3 \log(n)$$

Teniendo el $T(n)$, para demostrar por definición el BigOh se empieza planteando cuál puede ser, ya sea de forma intuitiva o por alguna de las reglas conocidas. En este caso, el orden parecería ser: $\mathcal{O}(n^3 \log(n))$.

Una vez planteado el Big-Oh posible, se aplica la definición para verificar si realmente el orden elegido corresponde con la función:

$$\exists k \in \mathbb{R}, T(n) \leq kF(n), \forall n \geq n_0 \tag{1.2.13}$$

Se reemplazan $T(n)$ y $F(n)$ por sus correspondientes valores. La función $F(n)$ es el orden elegido, $F(n) = n^3 \log(n)$:

$$2n^3 + 3n^3 \log(n) \leq kn^3 \log(n), \forall n \geq n_0 \tag{1.2.14}$$

Se pasa dividiendo $n^3 \log(n)$ hacia la izquierda:

$$\frac{2n^3 + 3n^3 \log(n)}{n^3 \log(n)} \leq k, \forall n \geq n_0 \tag{1.2.15}$$

$$\frac{2}{\log(n)} + 3 \leq k, \forall n \geq n_0, n_0 = 2 \tag{1.2.16}$$

Se puede ver que es posible encontrar un k fijo que, sin importar el valor tomado por n, la desigualdad sea siempre cierta.

Esto ocurre porque mientras n es más grande, $2/\log(n)$ es cada vez más chico. Por lo tanto, la función nunca va a tomar un valor mayor que algún k. En este ejemplo, $k = 5$ para que la desigualdad sea verdadera para todo $n \geq n_0$, con $n_0 = 2$.

El valor de n_0 no puede ser menor que 2 ya que con $n = 1$ se tendrá una división por cero.

El objetivo de la demostración es encontrar un k fijo de forma que se cumpla la desigualdad, para todo $n \geq n_0$, y lo encontramos. Por lo tanto, $T(n)$ es de orden $\mathcal{O}(n^3 \log(n))$.

1.2.7. Ejemplo 3, por definición Big-Oh en partes

Calcular y demostrar el orden de $T(n)$ por definición:

$$T(n) = 5n^3 + 2n^2 + n\log(n) \tag{1.2.17}$$

Una segunda manera de demostrar el orden a partir de un $T(n)$ es en partes. Se toma cada término por separado, y se demuestra el orden de cada uno. Si en todas las partes se cumple la desigualdad, quiere decir que se cumple en la función original.

De la misma forma que siempre, hay que elegir un orden para luego verificar si realmente se cumple. En este caso se elige:

$$\mathcal{O}(n^3)$$

Es importante que todos los términos se comparen **con el mismo orden**, de otra forma la demostración no será correcta.

Planteando los tres términos por separado se tiene:

$$T_1(n) = 5n^3 \tag{1.2.18}$$
$$T_2(n) = 2n^2 \tag{1.2.19}$$
$$T_3(n) = n\log(n) \tag{1.2.20}$$

Demostrando cada uno por separado:

$$\exists k_1, T_1(n) \leq k_1 F(n), \forall n \geq n_{1,0} \tag{1.2.21}$$
$$\exists k_2, T_2(n) \leq k_2 F(n), \forall n \geq n_{2,0} \tag{1.2.22}$$
$$\exists k_3, T_3(n) \leq k_3 F(n), \forall n \geq n_{3,0} \tag{1.2.23}$$

Reemplazando las tres desigualdades se tiene:

$$\exists k_1, 5n^3 \leq k_1 n^3, \forall n \geq n_1 \tag{1.2.24}$$
$$\frac{5n^3}{n^3} \leq k_1 \longrightarrow 5 \leq k_1 \tag{1.2.25}$$

$$\exists k_2, 2n^2 \leq k_2 n^3, \forall n \geq n_2 \tag{1.2.26}$$
$$\frac{2n^2}{n^3} \leq k_2 \longrightarrow \frac{2}{n} \leq k_2 \tag{1.2.27}$$

$$\exists k_3, n\log(n) \leq k_3 n^3, \forall n \geq n_3 \tag{1.2.28}$$
$$\frac{n\log(n)}{n^3} \leq k_3 \longrightarrow \frac{\log(n)}{n^2} \leq k_3 \tag{1.2.29}$$

En los tres casos se llega a que la desigualdad se cumple, siendo posible encontrar un $k_i \in \mathbb{R}$ fijo para cualquier valor positivo de n, mayor que algun n_i. Por lo tanto, $T(n)$ es de orden $\mathcal{O}(n^3)$.

1.2.8. Ejemplo 4, por definición.

¿Cuál es el orden $\mathcal{O}()$ de la siguiente función?

$$F(n) = n^{1/2} + n^{1/2}\log_4(n)$$

a) ¿Es de orden $\mathcal{O}(n^{1/2})$? Para que $F(n)$ sea $\mathcal{O}(n^{1/2})$ hace falta encontrar un k constante tal que se cumpla la siguiente desigualdad para todo $n \geq$ algún n_0.

$$n^{1/2} + n^{1/2}\log_4(n) \leq kn^{1/2} \tag{1.2.30}$$

Pasando dividiendo el $n^{1/2}$ hacia la izquierda queda:

$$\frac{n^{1/2}}{n^{1/2}} + \frac{n^{1/2}\log_4(n)}{n^{1/2}} \leq k \tag{1.2.31}$$
$$1 + \log_4(n) \leq k \tag{1.2.32}$$

Se concluye que es falso, porque a medida que n aumenta, la parte de la izquierda de la desigualdad $1 + \log_4(n)$ también aumenta, para todo $n \geq$ cualquier n_0, y no es posible encontrar un k constante. No es de orden $\mathcal{O}(n^{1/2})$.

b) ¿Es de orden $\mathcal{O}(\log_4(n))$? Para que $F(n)$ sea $\mathcal{O}(\log_4(n))$ hace falta encontrar un k constante tal que se cumpla la siguiente desigualdad para todo $n \geq$ algún n_0.

$$n^{1/2} + n^{1/2}\log_4(n) \leq k\log_4(n) \tag{1.2.33}$$

Pasando dividiendo el $\log_4(n)$ hacia la izquierda queda:

$$\frac{n^{1/2}}{\log_4(n)} + \frac{n^{1/2}\log_4(n)}{\log_4(n)} \leq k \tag{1.2.34}$$

$$\frac{n^{1/2}}{\log_4(n)} + n^{1/2} \leq k \tag{1.2.35}$$

Se concluye que es falso, porque a medida que n aumenta, la parte de la izquierda de la desigualdad $\frac{n^{1/2}}{\log_4(n)} + n^{1/2}$ también aumenta, para todo $n \geq$ cualquier n_0, y no es posible encontrar un k constante. No es de orden $\mathcal{O}(\log_4(n))$.

c) ¿Es de orden $\mathcal{O}(n^{1/2}\log_4(n))$? Para que $F(n)$ sea $\mathcal{O}(n^{1/2}\log_4(n))$ hace falta encontrar un k constante tal que se cumpla la siguiente desigualdad para todo $n \geq$ algún n_0.

$$n^{1/2} + n^{1/2}\log_4(n) \leq kn^{1/2}\log_4(n) \tag{1.2.36}$$

Pasando dividiendo el $n^{1/2}\log_4(n)$ hacia la izquierda queda:

$$\frac{n^{1/2}}{n^{1/2}\log_4(n)} + \frac{n^{1/2}\log_4(n)}{n^{1/2}\log_4(n)} \leq k \tag{1.2.37}$$

$$\frac{1}{\log_4(n)} + 1 \leq k \tag{1.2.38}$$

Se concluye que es verdadero, porque a medida que n aumenta, la parte de la izquierda de la desigualdad $\frac{1}{\log_4(n)} + 1$ va tomando cada vez valores más pequeños para todo $n \geq 2, n_0 = 2$. Por lo tanto, es posible encontrar el k constante y fijo que haga que se cumpla la desigualdad y $F(n)$ es de orden $\mathcal{O}(n^{1/2}\log_4(n))$.

1.3. Análisis de Algoritmos y Recurrencias

El análisis de algoritmos consiste en convertir un bloque de código o función escrita en algún lenguaje de programación a su equivalente versión que nos permita calcular su función de tiempo de ejecución $T(n)$.

Eso es, el objetivo es encontrar cual es el tiempo de ejecución a partir de la cantidad de elementos.

El objetivo de resolver una recurrencia es convertir la función en su forma explícita equivalente sin contener sumatorias ni llamadas recursivas. Para ello hace falta aplicar las definiciones previas de sumatorias, y el mecanismo de resolución de recurrencias.

1.3.1. Expresión constante

Cualquier expresión que no dependa de la cantidad de elementos a procesar se marcará como constante, $c_1, ..., c_k$.

1.3.2. Grupo de constantes

Cualquier grupo de constantes $c_1, ..., c_k$ se pueden agrupar en una única constante c, $c = (c_1 + ... + c_k)$.

1.3.3. Secuencias

Dadas dos o más sentencias una después de la otra sus tiempos se suman.

1.3.4. Condicionales

Dado un condicional *if*, el tiempo es el peor caso entre todos los caminos posibles de ejecución.

1.3.5. for y while

Los bloques *for* y *while* se marcan como la cantidad de veces que se ejecutan, expresado como una sumatoria desde 1 a esa cantidad. En el caso del *for* se usará como variable de la sumatoria el mismo indice del for. En el caso del *while* se usará una variable que no haya sido usada.

1.3.6. for y while, ejemplo

Se tiene el siguiente código:

```
void int f(int N) {
   int x = 1;
   while (x < N) {
      /* Algo de O(1)... */
      x = x * 2;
   }
   return x;
}
```

Tal como se describió, hace falta escribir el tiempo $T(n)$ como una sumatoria de 1 hasta la cantidad de veces que se ejecuta el *while*. La cantidad de veces que éste *while* se ejecuta se puede plantear de la siguiente forma:

1. El *while* se ejecuta mientras se cumple la condición $x < N$.
2. La variable x comienza en 1, en cada paso se multiplica su valor por 2, y la ejecución termina cuando x llega a N.
3. La cantidad de veces que se ejecuta el *while* corresponde con la cantidad de veces que la variable x se multiplica por 2.

Si decimos que la variable k representa la cantidad de veces que se multiplica la variable x por dos, la variable x entonces vale:

$$x = 1 \times 2 \times 2 \times ... \times 2 = 2^k \quad (1.3.1)$$

Además pedimos que x sea igual a N para que el while finalice:

$$x = N \quad (1.3.2)$$

Nos interesa entonces hallar el valor de k, a fin de hallar la cantida de veces que se ejecuta el *while*. Eso se logra facilmente aplicando propiedades de logaritmos:

$$\begin{aligned} x &= 2^k && (1.3.3) \\ x &= N && (1.3.4) \\ 2^k &= N && (1.3.5) \\ \log_2(2^k) &= \log_2 N && (1.3.6) \\ k &= \log_2 N && (1.3.7) \end{aligned}$$

Por lo tanto, el $T(n)$ del ejemplo anterior es:

$$T(n) = c_1 + \sum_{x=1}^{\log_2 N} (c_2) \tag{1.3.8}$$

1.3.7. Llamadas recursivas

Las llamadas recursivas se reemplazan por la definicion de tiempo recursiva. Las definiciones recursivas son funciones definidas por partes, teniendo por un lado el caso base y por el otro el caso recursivo.

1.3.8. Ejemplo 5, iterativo

```
1 void int sumar(int[] datos) {
2     int acumulado = 0;
3     for (int i=0; i<datos.length; i++) {
4         acumulado = acumulado + datos[i];
5     }
6     return acumulado;
7 }
```

Linea 2: c_1.

Linea 3: $\sum_{i=1}^{n}$.

Linea 4: c_2.

Linea 6: c_3.

Todo junto, el $T(n)$ de la función sumar es:

$$T(n) = c_1 + (\sum_{i=1}^{n} c_2) + c_3 \tag{1.3.9}$$

El siguiente paso es convertir el $T(n)$ a su versión explícita sin sumatorias. Aplicando las identidades de la sección anterior:

$$T(n) = c_1 + nc_2 + c_3 = c_4 + nc_2, c_4 = c_1 + c_3 \tag{1.3.10}$$

Por lo tanto, el $T(n)$ de la función sumar es:

$$T(n) = c_4 + nc_2 \tag{1.3.11}$$

¿Cuál es su orden de ejecución BigOh? $T(n)$ es $\mathcal{O}(n)$.

1.3.9. Ejemplo 6, recursivo

```
int rec1(int n){
  if (n <= 1)
    return 3;
  else
    return 1 + rec1(n-1) * rec1(n-1);
}
```

Las funciones recursivas se definen por partes, donde cada parte se resuelve de forma similar a un codigo iterativo:

$$T(n) = \begin{cases} c_1 & n \leq 1 \\ 2T(n-1) + c_2 & n > 1 \end{cases}$$

El objetivo es encontrar una versión explícita de la recurrencias. Se comienza por escribir las primeras definiciones parciales:

$$T(n) = 2T(n-1) + c_2, n > 1 \quad (1.3.12)$$
$$T(n-1) = 2T(n-2) + c_2, n-1 > 1 \quad (1.3.13)$$
$$T(n-2) = 2T(n-3) + c_2, n-2 > 1 \quad (1.3.14)$$

Segundo, se expande cada término tres ó cuatro veces, las que sean necesarias para descubrir la forma en la que los diferentes términos van cambiando:

$$T(n) = 2T(n-1) + c_2 \quad (1.3.15)$$
$$T(n) = 2(2T(n-2) + c_2) + c_2 \quad (1.3.16)$$
$$T(n) = 2(2(2T(n-3) + c_2) + c_2) + c_2 \quad (1.3.17)$$
$$T(n) = 2(2^2T(n-3) + 2c_2 + c_2) + c_2 \quad (1.3.18)$$
$$T(n) = 2^3T(n-3) + 2^2c_2 + 2c_2 + c_2, \forall n-3 \geq 1 \quad (1.3.19)$$

La definición recursiva continúa mientras el $n > 1$, por lo que en k pasos tenemos la expresión general del paso k:

$$T(n) = 2^kT(n-k) + \sum_{i=0}^{k-1} 2^i c_2 \quad (1.3.20)$$

La llamada recursiva finaliza cuando $n - k = 1$, por lo tanto:

$$n - k = 1 \tag{1.3.21}$$
$$k = n - 1 \tag{1.3.22}$$
$$T(n) = 2^{n-1}T(1) + \sum_{i=0}^{n-1-1} 2^i c_2 \tag{1.3.23}$$
$$T(n) = 2^{n-1}c_1 + \sum_{i=0}^{n-2} 2^i c_2 \tag{1.3.24}$$
$$T(n) = 2^{n-1}c_1 + c_2 \sum_{i=0}^{n-2} 2^i \tag{1.3.25}$$
$$T(n) = 2^{n-1}c_1 + c_2(2^{n-2+1} - 1) \tag{1.3.26}$$
$$T(n) = 2^{n-1}c_1 + 2^{n-1}c_2 - c_2 \tag{1.3.27}$$
$$T(n) = 2^{n-1}(c_1 + c_2) - c_2 \tag{1.3.28}$$

¿Cuál es su orden de ejecución BigOh? $T(n)$ es $\mathcal{O}(2^n)$.

1.3.10. Ejemplo 7, recursivo

$$T(n) = \begin{cases} c & n = 1 \\ 2n + T(n/2) & n \geq 2 \end{cases}$$

Se asume que n es potencia entera de 2.

$$\sum_{k=0}^{n} \frac{1}{2^k} = 2 - \frac{1}{2^n}$$

Definiciones parciales:

$$T(n) = 2n + T(n/2)$$
$$T(\frac{n}{2}) = \frac{2n}{2} + T\left(\frac{n}{2^2}\right)$$
$$T(\frac{n}{2^2}) = \frac{2n}{2^2} + T\left(\frac{n}{2^3}\right)$$
$$T(\frac{n}{2^3}) = \frac{2n}{2^3} + T\left(\frac{n}{2^4}\right)$$

Desarrollo:

$$T(n) = 2n + T(n/2) \tag{1.3.29}$$

$$T(n) = 2n + \left(\frac{2n}{2} + T\left(\frac{n}{2^2}\right)\right) \tag{1.3.30}$$

$$T(n) = 2n + \frac{2n}{2} + \left(\frac{2n}{2^2} + T\left(\frac{n}{2^3}\right)\right) \tag{1.3.31}$$

$$T(n) = 2n + \frac{2n}{2} + \frac{2n}{2^2} + \left(\frac{2n}{2^3} + T\left(\frac{n}{2^4}\right)\right) \tag{1.3.32}$$

$$T(n) = \sum_{i=0}^{3} \frac{2n}{2^i} + T\left(\frac{n}{2^4}\right) \tag{1.3.33}$$

$$T(n) = 2n \sum_{i=0}^{3} \frac{1}{2^i} + T\left(\frac{n}{2^4}\right) \tag{1.3.34}$$

Paso general k:

$$T(n) = 2n \sum_{i=0}^{k-1} \frac{1}{2^i} + T\left(\frac{n}{2^k}\right) \tag{1.3.35}$$

Caso base:

$$\frac{n}{2^k} = 1 \rightarrow n = 2^k \rightarrow \log n = k$$

$$T(n) = 2n \sum_{i=0}^{\log n-1} \frac{1}{2^i} + T\left(\frac{n}{2^{\log n}}\right) \tag{1.3.36}$$

$$T(n) = 2n \sum_{i=0}^{\log n-1} \frac{1}{2^i} + c \tag{1.3.37}$$

$$T(n) = 2n \left(2 - \frac{1}{2^{\log n-1}}\right) + c \tag{1.3.38}$$

$$T(n) = 4n - \frac{2n}{2^{\log n-1}} + c \tag{1.3.39}$$

$$\tag{1.3.40}$$

$$T(n) = 4n - \frac{2n}{(2^{\log n})/2} + c \quad (1.3.41)$$

$$T(n) = 4n - \frac{2n}{(n)/2} + c \quad (1.3.42)$$

$$T(n) = 4n - \frac{4n}{n} + c \quad (1.3.43)$$

$$T(n) = 4n - 4 + c \quad (1.3.44)$$

1.3.11. Ejemplo 8, Ejercicio 12.4

$$T(n) = \begin{cases} 1 & n = 1 \\ 8T(\frac{n}{2}) + n^3 & n \geq 2 \end{cases}$$

Soluciones parciales:

$$T(n) = 8T(\frac{n}{2}) + n^3 \quad (1.3.45)$$

$$T(\frac{n}{2}) = 8T(\frac{n}{4}) + (\frac{n}{2})^3 \quad (1.3.46)$$

$$T(\frac{n}{4}) = 8T(\frac{n}{8}) + (\frac{n}{4})^3 \quad (1.3.47)$$

$$T(\frac{n}{8}) = 8T(\frac{n}{16}) + (\frac{n}{8})^3 \quad (1.3.48)$$

Desarrollo:

$$T(n) = 8T(\frac{n}{2}) + n^3 \quad (1.3.49)$$

$$T(n) = 8(8T(\frac{n}{4}) + (\frac{n}{2})^3) + n^3 \quad (1.3.50)$$

$$T(n) = 8(8(8T(\frac{n}{8}) + (\frac{n}{4})^3) + (\frac{n}{2})^3) + n^3 \quad (1.3.51)$$

$$T(n) = 8(8(8(8T(\frac{n}{16}) + (\frac{n}{8})^3) + (\frac{n}{4})^3) + (\frac{n}{2})^3) + n^3 \quad (1.3.52)$$

$$T(n) = 8(8(8^2T(\frac{n}{16}) + 8(\frac{n}{8})^3 + (\frac{n}{4})^3) + (\frac{n}{2})^3) + n^3 \quad (1.3.53)$$

$$T(n) = 8(8^3T(\frac{n}{16}) + 8^2(\frac{n}{8})^3 + 8(\frac{n}{4})^3 + (\frac{n}{2})^3) + n^3 \quad (1.3.54)$$

$$T(n) = 8^4T(\frac{n}{16}) + 8^3(\frac{n}{8})^3 + 8^2(\frac{n}{4})^3 + 8(\frac{n}{2})^3 + n^3 \quad (1.3.55)$$

$$n^3 = 8^0(\frac{n}{2^0})^3 \quad (1.3.56)$$

$$T(n) = 8^4T(\frac{n}{2^4}) + 8^3(\frac{n}{2^3})^3 + 8^2(\frac{n}{2^2})^3 + 8^1(\frac{n}{2^1})^3 + 8^0(\frac{n}{2^0})^3 \quad (1.3.57)$$

$$T(n) = 8^4T(\frac{n}{2^4}) + \sum_{i=0}^{3} 8^i(\frac{n}{2^i})^3 \quad (1.3.58)$$

Caso k:

$$T(n) = 8^kT(\frac{n}{2^k}) + \sum_{i=0}^{k-1} 8^i(\frac{n}{2^i})^3 \quad (1.3.59)$$

Caso base:

$$\frac{n}{2^k} = 1 \rightarrow n = 2^k \rightarrow \log n = \log 2^k = k$$

$$k = \log n \quad (1.3.60)$$

$$T(n) = 8^{\log n}T(\frac{n}{2^{\log n}}) + \sum_{i=0}^{\log n-1} 8^i(\frac{n}{2^i})^3 \quad (1.3.61)$$

$$T(n) = 8^{\log n}T(\frac{n}{n}) + \sum_{i=0}^{\log n-1} 8^i(\frac{n^3}{(2^i)^3}) \quad (1.3.62)$$

$$T(n) = 8^{\log n}T(1) + \sum_{i=0}^{\log n-1} (\frac{(2^3)^i n^3}{2^{3i}}) \quad (1.3.63)$$

$$T(n) = 8^{\log n}T(1) + \sum_{i=0}^{\log n-1} (\frac{2^{3i} n^3}{2^{3i}}) \quad (1.3.64)$$

$$T(n) = 8^{\log n}T(1) + \sum_{i=0}^{\log n-1} (n^3) \quad (1.3.65)$$

$$T(n) = 8^{\log n} + n^3 \log n \quad (1.3.66)$$

$$T(n) = (2^3)^{\log n} + n^3 \log n \quad (1.3.67)$$

$$T(n) = 2^{3\times\log n} + n^3 \log n \quad (1.3.68)$$

$$T(n) = (2^{(\log n)})^3 + n^3 \log n \quad (1.3.69)$$

$$T(n) = n^3 + n^3 \log n \quad (1.3.70)$$

1.4. Otros ejemplos

1.4.1. Ejemplo 9

Sea el siguiente programa,

a. Calcular su $T(n)$, detallando los pasos seguidos para llegar al resultado.

b. Calcular su $\mathcal{O}(f(n))$ justificando usando la definición de BigOh.

```
int uno(int n){
  for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= i; j++) {
      algo_de_O(1);
    }
  }
  if (n > 1)
    for (int i = 1; i <= 4; i++)
      uno(n / 2);
}
```

Planteo de la recurrencia

$$T(n) = \begin{cases} c_1 & n = 1 \\ \sum_{i=1}^{n} \sum_{j=1}^{i} c_2 + 4T(\frac{n}{2}) & n \geq 2 \end{cases}$$

Eliminando las sumatorias tenemos:

$$\sum_{i=1}^{n} \sum_{j=1}^{i} c_2 = \sum_{i=1}^{n} i c_2 = c_2 \sum_{i=1}^{n} i = \frac{c_2 n(n+1)}{2} = \frac{c_2}{2}(n^2 + n) \tag{1.4.1}$$

$$T(n) = \begin{cases} c_1 & n = 1 \\ \frac{c_2}{2}(n^2 + n) + 4T(\frac{n}{2}) & n \geq 2 \end{cases}$$

Desarrollo de la recurrencia:

$$T(n) = \frac{c_2}{2}(n^2 + n) + 4T(\frac{n}{2}) \tag{1.4.2}$$

$$= \frac{c_2}{2}(n^2+n) + 4\left(\frac{c_2}{2}\left(\left(\frac{n}{2}\right)^2 + \left(\frac{n}{2}\right)\right) + 4T\left(\frac{n}{2^2}\right)\right) \tag{1.4.3}$$

$$= \frac{c_2}{2}(n^2+n) + 4\frac{c_2}{2}\left(\left(\frac{n}{2}\right)^2 + \left(\frac{n}{2}\right)\right) + 4^2T\left(\frac{n}{2^2}\right) \tag{1.4.4}$$

$$\begin{aligned} &= \frac{c_2}{2}(n^2+n) + 4\frac{c_2}{2}\left(\left(\frac{n}{2}\right)^2 + \left(\frac{n}{2}\right)\right) + \\ &4^2\left(\frac{c_2}{2}\left(\left(\frac{n}{2^2}\right)^2 + \left(\frac{n}{2^2}\right)\right) + 4T\left(\frac{n}{2^3}\right)\right) \end{aligned} \tag{1.4.5}$$

$$\begin{aligned} &= \frac{c_2}{2}(n^2+n) + 4\frac{c_2}{2}\left(\left(\frac{n}{2}\right)^2 + \left(\frac{n}{2}\right)\right) + \\ &4^2\frac{c_2}{2}\left(\left(\frac{n}{2^2}\right)^2 + \left(\frac{n}{2^2}\right)\right) + 4^3T\left(\frac{n}{2^3}\right) \end{aligned} \tag{1.4.6}$$

$$\begin{aligned} &= \frac{c_2}{2}(n^2+n) + 4\frac{c_2}{2}\left(\left(\frac{n}{2}\right)^2 + \left(\frac{n}{2}\right)\right) + \\ &4^2\frac{c_2}{2}\left(\left(\frac{n}{2^2}\right)^2 + \left(\frac{n}{2^2}\right)\right) + \\ &4^3\left(\frac{c_2}{2}\left(\left(\frac{n}{2^3}\right)^2 + \left(\frac{n}{2^3}\right)\right) + 4T\left(\frac{n}{2^4}\right)\right) \end{aligned} \tag{1.4.7}$$

$$\begin{aligned} &= \frac{c_2}{2}(n^2+n) + 4\frac{c_2}{2}\left(\left(\frac{n}{2}\right)^2 + \left(\frac{n}{2}\right)\right) + \\ &4^2\frac{c_2}{2}\left(\left(\frac{n}{2^2}\right)^2 + \left(\frac{n}{2^2}\right)\right) + \\ &4^3\frac{c_2}{2}\left(\left(\frac{n}{2^3}\right)^2 + \left(\frac{n}{2^3}\right)\right) + 4^4T\left(\frac{n}{2^4}\right) \end{aligned} \tag{1.4.8}$$

$$= \frac{c_2}{2}\sum_{w=0}^{3}\left(4^w\left(\left(\frac{n}{2^w}\right)^2 + \left(\frac{n}{2^w}\right)\right)\right) + 4^4T\left(\frac{n}{2^4}\right) \tag{1.4.9}$$

$$= \frac{c_2}{2}\sum_{w=0}^{3}\left(4^w\left(\frac{n^2}{(2^w)^2}\right) + 4^w\left(\frac{n}{2^w}\right)\right) + 4^4T\left(\frac{n}{2^4}\right) \tag{1.4.10}$$

$$= \frac{c_2}{2}\sum_{w=0}^{3}\left(2^{2w}\left(\frac{n^2}{2^{2w}}\right) + 2^{2w}\left(\frac{n}{2^w}\right)\right) + 4^4T\left(\frac{n}{2^4}\right) \tag{1.4.11}$$

$$= \frac{c_2}{2}\sum_{w=0}^{3}\left(\left(n^2\right) + 2^w\left(n\right)\right) + 4^4T\left(\frac{n}{2^4}\right) \tag{1.4.12}$$

$$T(n) = \frac{c_2}{2}\sum_{w=0}^{3} n^2 + \frac{c_2}{2}\sum_{w=0}^{3} 2^w n + 4^4 T\left(\frac{n}{2^4}\right) \tag{1.4.13}$$

Paso general k:

$$T(n) = \frac{c_2}{2}\sum_{w=0}^{k-1} n^2 + \frac{c_2}{2}\sum_{w=0}^{k-1} 2^w n + 4^k T\left(\frac{n}{2^k}\right) \tag{1.4.14}$$

Paso base:

$$\frac{n}{2^k} = 1 \tag{1.4.15}$$
$$n = 2^k \tag{1.4.16}$$
$$\log_2 n = \log_2(2^k) \tag{1.4.17}$$
$$\log_2 n = k \tag{1.4.18}$$

$$T(n) = \frac{c_2}{2}\sum_{w=0}^{\log_2(n)-1} n^2 + \frac{c_2}{2}\sum_{w=0}^{\log_2(n)-1} 2^w n + 4^{\log_2 n} T\left(\frac{n}{2^{\log_2 n}}\right) \tag{1.4.19}$$
$$= \frac{c_2}{2} n^2 \log_2(n) + \frac{c_2}{2}(\log_2(n)+1)n + 4^{\log_2 n} T\left(\frac{n}{2^{\log_2 n}}\right) \tag{1.4.20}$$
$$4^{\log_2 n} = (2^2)^{\log_2 n} = (2^{\log_2 n})^2 = n^2 \tag{1.4.21}$$
$$= \frac{c_2}{2} n^2 \log_2(n) + \frac{c_2}{2}(\log_2(n)+1)n + n^2 T(1) \tag{1.4.22}$$
$$T(n) = \frac{c_2}{2} n^2 \log_2(n) + \frac{c_2}{2} n \log_2(n) + \frac{c_2}{2} n + c_1 n^2 \tag{1.4.23}$$

Demostración del orden $\mathcal{O}(f(n))$

Se elige el término de mayor orden: $\frac{c_2}{2} n^2 \log_2(n)$
$f(n)$ es de $\mathcal{O}(g(n))$ si y solo si $\exists w$ tal que $f(n) \leq w g(n), \forall n \geq n_0$.
¿$T(n)$ es de $\mathcal{O}(n^2 \log_2(n))$?

$$\frac{c_2}{2} n^2 \log_2(n) + \frac{c_2}{2} n \log_2(n) + \frac{c_2}{2} n + c_1 n^2 \leq w n^2 \log_2(n) \tag{1.4.24}$$
$$\frac{c_2 n^2 \log_2(n)}{2n^2 \log_2(n)} + \frac{c_2 n \log_2(n)}{2n^2 \log_2(n)} + \frac{c_2 n}{2n^2 \log_2(n)} + \frac{c_1 n^2}{n^2 \log_2(n)} \leq w \tag{1.4.25}$$
$$\frac{c_2}{2} + \frac{c_2}{2n} + \frac{c_2}{2n \log_2(n)} + \frac{c_1}{\log_2(n)} \leq w \tag{1.4.26}$$

Dado que la parte izquierda de la desigualdad es una función decreciente, con $n_0 = 2$ se tiene que se cumple la definición de BigOh, donde $T(n) \leq wn^2 \log_2(n)$, $\forall n \geq 2$.

$$\frac{c_2}{2} + \frac{c_2}{4} + \frac{c_2}{4\log_2(2)} + \frac{c_1}{\log_2(2)} = \frac{c_2}{2} + \frac{c_2}{4} + \frac{c_2}{4} + \frac{c_1}{1} = c_2 + c_1 = w \tag{1.4.27}$$

Por lo tanto se demuestra que el $T(n)$ es efectivamente $\mathcal{O}(n^2 \log_2(n))$.

1.4.2. Ejemplo 10

Sea el siguiente programa,

a. Calcular su $T(n)$, detallando los pasos seguidos para llegar al resultado.

b. Calcular su $\mathcal{O}(f(n))$ justificando usando la definición de BigOh.

```
public static void f(int n) {
  int sum=0, sum1=0, sum2=0;
  for (double i=1; i<= n; i += 0.5) {
    ++sum1;
    for (double j=1; j <= sum1; j++) {
      sum2++;
      for (double k=j; k <= sum1; k++) {
        sum += sum1 + sum2;
      }
    }
  }
  return sum;
}
```

Antes de empezar

Para plantear inicialmente este ejercicio hace falta analizar el programa. Una estrategia interesante para resolver este tipo de ejercicios consiste en reescribir los límites de los bucles, a fin de conseguir un programa que se ejecute la misma cantidad de veces que el original y calcule los mismos valores con el mismo resultado, pero que sea más fácil para analizar.

Por ejemplo, los programas `F1` y `F2` son equivalentes en cuanto al valor calculado y cantidad de veces que se ejecuta cada bucle, solo varían en sus constantes:

```
public static void F1(int n) {
  int x = 0;
  for (int i = 10; i<= 10 * n; i += 10) {
    x += algo_de_O1(i);
  }
  return x;
}
```

```
public static void F2(int n) {
  int x = 0;
  for (int i = 1; i <= n; i++) {
    x += algo_de_O1(i*10);
  }
  return x;
}
```

Al momento de escribir el tiempo de ejecución de los programas, se pueden tener:

1. Sumatorias que sus variables tomen los mismos valores que los bucles del programa.

2. Sumatorias que tengan la misma cantidad de términos que la cantidad de veces que se ejecuta cada bucle del programa.

Estas dos estrategias son equivalentes entre sí. Por esto, el tiempo de ejecución de ambas funciones `F1` y `F2` anteriores es:

$$F(n) = C_1 + C_2 n + C_3 \tag{1.4.28}$$

Donde C_1, C_2, C_3 son constantes que su valor depende de la versión del programa se está analizando, ya sea `F1` o `F2`.

Ejercicio

A partir de esto podemos decir:

- El `for` con índice i se ejecuta `2 * n - 1` veces.

- La variable `sum1` cuenta la cantidad de veces que se ejecuta el primer `for`. Su valor comienza en 1 y termina en `2 * n - 1`.

- El `for` con índice j se ejecuta tantas veces como vale la variable `sum1`. Dado que el valor de la variable `sum1` va cambiando a medida que se ejecuta el programa, no es correcto decir que éste for se ejecuta `2 * n` veces, porque `n` siempre vale lo mismo pero `sum1` no.
- El `for` con índice k se ejecuta `sum1 - j + 1` veces.
- Dado que el índice i del primer for no se utiliza sino solo para contar (y no aparece en ningún otro lugar del programa) podemos cambiar el primer `for` (con índice i) por esto otro:

```
for (int i=1; i <= (2 * n - 1); i++) { ...
```

  Con este cambio conseguimos que la variable i tome el mismo valor que la variable `sum1`, manteniendo que éste `for` se ejecute la misma cantidad de veces que el `for` original.

A partir de este análisis, podemos definir la función de tiempo:

$$T(n) = \sum_{i=1}^{2n-1} \sum_{j=1}^{i} \sum_{k=j}^{i} c_1 \tag{1.4.29}$$

$$T(n) = c_1 \sum_{i=1}^{2n-1} \sum_{j=1}^{i} (i - j + 1) \tag{1.4.30}$$

$$T(n) = c_1 \sum_{i=1}^{2n-1} \left(\sum_{j=1}^{i} i - \sum_{j=1}^{i} j + \sum_{j=1}^{i} 1 \right) \tag{1.4.31}$$

$$T(n) = c_1 \sum_{i=1}^{2n-1} \left(i^2 - \frac{i(i+1)}{2} + i \right) \tag{1.4.32}$$

$$T(n) = c_1 \sum_{i=1}^{2n-1} \left(i^2 - \frac{1}{2}i^2 - \frac{1}{2}i + i \right) \tag{1.4.33}$$

$$T(n) = c_1 \sum_{i=1}^{2n-1} \left(\frac{1}{2}i^2 + \frac{1}{2}i \right) \tag{1.4.34}$$

$$T(n) = c_1 \frac{1}{2} \left(\sum_{i=1}^{2n-1} i^2 + \sum_{i=1}^{2n-1} i \right) \tag{1.4.35}$$

$$\sum_{i=0}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$$

$$\sum_{i=1}^{2n-1} i^2 = \left(\sum_{i=1}^{2n} i^2\right) - (2n)^2 \tag{1.4.36}$$

$$T(n) = c_1 \frac{1}{2}\left(\frac{(2n)((2n)+1)(2(2n)+1)}{6} - (2n)^2 + \frac{(2n-1)(2n)}{2}\right) \tag{1.4.37}$$

$$T(n) = c_1 \frac{1}{2}\left(\frac{(4n^2+2n)(4n+1)}{6} - 4n^2 + \frac{4n^2-2n}{2}\right) \tag{1.4.38}$$

$$T(n) = c_1 \frac{1}{2}\left(\frac{16n^3+4n^2+8n^2+2n}{6} - 4n^2 + \frac{4n^2-2n}{2}\right) \tag{1.4.39}$$

$$T(n) = c_1 \frac{1}{2}\left(\frac{16}{6}n^3 + \frac{4}{6}n^2 + \frac{8}{6}n^2 + \frac{2}{6}n - 4n^2 + \frac{4}{2}n^2 - \frac{2}{2}n\right) \tag{1.4.40}$$

$$T(n) = c_1 \frac{1}{2}\left(\frac{16}{6}n^3 - \frac{2}{3}n\right) \tag{1.4.41}$$

$$T(n) = c_1 \frac{4}{3}n^3 - c_1 \frac{2}{3}n \tag{1.4.42}$$

Demostración del orden $\mathcal{O}(f(n))$

Se elige el término de mayor orden: $g(n) = n^3$
$f(n)$ es de $\mathcal{O}(g(n))$ si y solo si $\exists w$ tal que $f(n) \leq wg(n)$, $\forall n \geq n_0$.
¿$T(n)$ es de $\mathcal{O}(n^3)$?

$$c_1 \frac{4}{3}n^3 - c_1 \frac{2}{3}n \leq wn^3 \; \forall n \geq n_0 \tag{1.4.43}$$

$$c_1 \frac{4n^3}{3n^3} - c_1 \frac{2n}{3n^3} \leq w \tag{1.4.44}$$

$$c_1 \frac{4}{3} - c_1 \frac{2}{3n^2} \leq w \tag{1.4.45}$$

Dado que la parte izquierda de la desigualdad es una función decreciente, con $n_0 = 1$ se tiene que se cumple la definición de BigOh, donde $T(n) \leq wn^3$, $\forall n \geq 1$.

$$w = c_1 \frac{4}{3} - c_1 \frac{2}{3} = \frac{2}{3}c_1 \tag{1.4.46}$$

Por lo tanto se demuestra que el $T(n)$ es efectivamente $\mathcal{O}(n^3)$.

1.5. Identidades de sumatorias y logaritmos

$$\sum_{i=1}^{n} c = nc \tag{1.5.1}$$

$$\sum_{i=k}^{n} c = (n-k+1)c \tag{1.5.2}$$

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2} \tag{1.5.3}$$

$$\sum_{i=0}^{n} i^2 = \frac{n(n+1)(2n+1)}{6} \tag{1.5.4}$$

$$\sum_{i=0}^{n} i^3 = \left(\frac{n(n+1)}{2}\right)^2 \tag{1.5.5}$$

$$\sum_{i=0}^{n} i^4 = \frac{n(n+1)(2n+1)(3n^2+3n-1)}{30} \tag{1.5.6}$$

$$\sum_{i=k}^{n} f(i) = \sum_{i=1}^{n} f(i) - \sum_{i=1}^{k-1} f(i) \tag{1.5.7}$$

$$\sum_{i=0}^{n} 2^i = 2^{n+1} - 1 \tag{1.5.8}$$

$$\sum_{i=0}^{n} a^i = \frac{1-a^{n+1}}{1-a} \tag{1.5.9}$$

$$\log_b a = \frac{log_c a}{\log_c b} \tag{1.5.10}$$

$$\log \frac{a}{b} = \log a - \log b \tag{1.5.11}$$

$$\log(a \times b) = \log a + \log b \tag{1.5.12}$$

$$\log_b(b^a) = b^{\log_b a} = a \tag{1.5.13}$$

$$\left(a^b\right)^c = a^{bc} = (a^c)^b \tag{1.5.14}$$

Capítulo 2

Listas

2.1. Los Pares del Faraón

Entre todas sus fascinaciones, al Faraón le atrae la idea de ver qué grupos de objetos a su alrededor se pueden dividir en dos grupos de igual tamaño.

Como Ingeniero/a del Faraón, es tu tarea escribir un programa en Java que, a partir de una lista que representa la cantidad de elementos de cada grupo, le permita al Faraón descubrir cuántos grupos se pueden dividir en dos sin tener un elemento sobrante (ni tener que dividir el elemento sobrante por la mitad).

```
int contar(ListaDeEnteros L) { ... }
```

2.1.1. Explicación del problema

Se tiene una lista de números enteros, y hay que devolver la cantidad de números pares que aparecen en la lista.

El objetivo de este ejercicio es poner en práctica los conceptos de abstración, teniendo un tipo abstracto que representa una secuencia de elementos sin importar de qué forma se representa internamente la estructura.

Es importante tener en cuenta que las listas se recorren con los métodos presentes en la práctica, sin acceder directamente al arreglo o nodos interiores.

2.1.2. Implementación Java

```
int contar(ListaDeEnteros L) {
  int cantidad = 0;

  L.comenzar();
  while (!L.fin()) {
    if ((L.elemento() % 2) == 0)
      cantidad++;
    L.proximo();
  }

  return cantidad;
}
```

2.2. PilaMin

(a) Implementar una clase *PilaDeEnteros* donde todas las operaciones sean de tiempo constante $\mathcal{O}(1)$.

(b) Implementar una subclase *PilaMin* que permita obtener el mínimo elemento presente en la estructura, también en tiempo constante $\mathcal{O}(1)$.

```
class PilaDeEnteros {
  public void poner(int e) {...}
  public int sacar() {...}
  public int tope() {...}
  public boolean esVacia() {...}
}

class PilaMin extends PilaDeEnteros {
  public int min() {...}
}
```

2.2.1. Explicación del problema

La implementación de *PilaDeEnteros* utiliza internamente una *ListaDeEnteros* para almacenar los elementos de la estructura.

De esta forma, la lista se utiliza agregando y eliminando de la pila siempre en el mismo extremo, ya sea siempre al comienzo de la estructura, o siempre al final.

Sim embargo, dependiendo de la implementación concreta de *ListaDeEnteros* elegida, la eficiencia será diferente. Utilizando una *ListaDeEnterosConArreglos*, para insertar al comienzo del arreglo es necesario hacer el corrimiento de todos los elementos de la estructura, teniendo orden lineal $\mathcal{O}(n)$, con n en tamaño de la estructua.

Algo similar ocurre en una *ListaDeEnterosEnlazada*, donde para insertar al final de la estructura es necesario recorrerla desde el comienzo para hacer los enganches de los nodos al final. Insertar un elemento a final de una lista enlazada simple es también una operacion de orden $\mathcal{O}(n)$.

De todas formas, es posible conseguir acceso de tiempo constante en una lista. En un arreglo, insertar y eliminar un elemento al final es una operación de tiempo constante, ya que sin importar la cantidad de elementos ni la posición a la que se accede, se puede acceder directamente y no hace falta hacer ningún corrimiento de elementos.

En una lista enlazada simple, insertar y eliminar en el comienzo es también de tiempo constante, ya que solo hace falta hacer los enganches del primer nodo de la estructura junto con la referencia al primer elemento, sin importar cuántos elementos contiene la estructura.

2.2.2. Implementación Java - Versión con Lista Enlazada

```
class PilaDeEnteros {
  private ListaDeEnterosEnlazada L = new
     ListaDeEnterosEnlazada();

  public void poner(int e) {
    L.comenzar();
    L.agregar(e); /* Tiempo cte. */
  }

  public int sacar() {
    int x;
    L.comenzar();
    x = L.elemento();
    L.eliminar(); /* Tiempo cte. */
  }

  public int tope() {
    L.comenzar();
    return L.elemento(); /* Tiempo cte. */
  }

  public boolean esVacia() {
    return (L.tamanio() == 0); /* Tiempo cte. */
  }
}
```

2.2.3. PilaMin, menor elemento en tiempo $\mathcal{O}(1)$

La operación para obtener el mínimo elemento de la estructura también debe ser de tiempo constante, por lo que recorrer la lista cada vez que se consulta el mínimo no es correcto.

Tampoco es correcto agregar una variable que acumule el mínimo entero agregado, porque si bien funciona mientras se le agregan elementos a la pila,

cuando se elimina un elemento de la pila ya pierde su valor. Por ejemplo:

```
class PilaMin extends PilaDeEnteros {
  int minelem = Integer.MAX_VALUE;

  public void poner(int e) {
    /* VERSION INCORRECTA */
    if (e < minelem) minelem = e;
    super.poner(e);
  }

  public int min() {
    /* VERSION INCORRECTA */
    return minelem;
  }
}
```

Es posible obtener el mínimo elemento en tiempo constante sin tener que recorrer la lista. A medida que se van agregando elementos, el nuevo menor es el menor elemento entre el menor actual y el nuevo elemento que se está agregando. Pero cuando se eliminan elementos, el menor debe pasar a ser el que estaba antes de agregarlo a la pila.

Por lo tanto, se puede crear una segunda pila que contenga la misma cantidad de elementos y en la que se vaya apilando el menor en cada paso. Entonces, el menor será el que esté en el tope de la segunda pila. Cuando se elimina un elemento de la estructura tambien es necesario eliminarlo de la pila de minimos.

```java
/* Version final correcta. */
class PilaMin extends PilaDeEnteros {
  private static int MIN(int a, int b) {
    return (a < b) ? a : b;
  }

  PilaDeEnteros minimos = new PilaDeEnteros();

  public void poner(int e) {
    if (esVacia())
      minimos.poner(e);
    else
      minimos.poner(MIN(e, minimos.tope()));
    super.poner(e);
  }

  public int sacar() {
    minimos.sacar();
    return super.sacar();
  }

  public int min() {
    return minimos.tope();
  }
}
```

Capítulo 3

Árboles binarios y Generales

3.1. Evaluar árbol de expresión

Dado un árbol de expresión, implementar en Java una función que calcule el valor de la expresión. El árbol ya se encuentra construido, es de `String` y la función será:

```
float evaluar(ArbolBinario<String> a) { ... }
```

Ejemplo: Dado el árbol 2 + 3 * 4, la función deberá devolver el valor 14.

3.1.1. Explicación del problema

El árbol es de `String`s porque hay diferentes tipos de nodos: operadores, y valores. Los operadores pueden ser $+, -, \times, /$, y los valores son números.

Dado que cualquier subarbol de un árbol de expresión es también un árbol de expresión, la forma más fácil y directa de resolverlo es de forma recursiva.

Si el árbol está vacío se dispara un `NullPointerException` en la primer llamada, pero dado que el árbol ya viene correctamente construido como un árbol de expresión, todos los nodos operadores tendrán dos hijos, y los nodos de valores serán hojas.

3.1.2. Implementación Java

```java
float evaluar(ArbolBinario<String> A) {
  String d = A.getDatoRaiz();
  ArbolBinario<String> I = A.getHijoIzquierdo();
  ArbolBinario<String> D = A.getHijoDerecho();

  if (d.equals("+"))
    return evaluar(I) + evaluar(D);
  if (d.equals("-"))
    return evaluar(I) - evaluar(D);
  if (d.equals("*"))
    return evaluar(I) * evaluar(D);
  if (d.equals("/"))
    return evaluar(I) / evaluar(D);

  return Float.parseFloat(d);
}
```

3.2. Mínimo del Nivel

Implementar en Java una función que dado un árbol binario de enteros y un número de nivel, permita calcular el mínimo valor del nivel indicado. Considere la raíz como nivel cero.

```
int calcularMinimo(ArbolBinario<Integer> a, int
    numeroNivel) { ... }
```

3.2.1. Explicación del problema

Hay dos formas diferentes de encarar este problema: de forma recursiva, y de forma iterativa. Ambas son simples de entender y correctas, dando el resultado esperado.

En la forma recursiva hace falta pasar por parámetro el árbol y el numero de nivel actual. Cuando se llega al nivel esperado hay que devolver el valor actual, y cuando se regresa de la recursión hay que comparar los minimos entre los hijos izquierdo y derecho del nodo actual.

En la forma iterativa hace falta recorrer por niveles el arbol, contando el nivel actual con la marca de `null`. Cuando se llega al nivel esperado hay que buscar entre todos los valores desencolados el mínimo de ellos. Una vez que se terminaron los elementos o se llegó a un nivel más que el buscado hay que cortar la búsqueda.

En ambos casos, hace falta verificar que el nivel realmente exista. Una forma de transmitirlo es devolviendo `null` cuando no se encontró el valor. Esto solo se puede cuando se usa `Integer` como tipo de datos; usando `int` como tipo de datos de retorno no es posible devolver `null`.

En el caso de usar `int` se puede usar la constante `Integer.MAX_VALUE` para indicar que el valor no se encontró.

En Java, el operador ternario signo de pregunta "`?:`" es un `if` reducido. La parte de la izquierda es la condición, cuando es `true` se devuelve el valor del medio, y cuando es `false` se devuelve lo que hay a la derecha del dos puntos.

Por ejemplo, la siguiente función devuelve el máximo entre los valores recibidos por parámetro:

```
int obtenerMaximo(int A, int B) {
  return (A > B) ? A : B;
}
```

3.2.2. Implementación Java - Recursivo con `Integer`

```
Integer calcularMinimo(ArbolBinario<Integer> A, int
    numeroNivel) {
  if (esVacio(A))
    return null;
  if (numeroNivel == 0)
    return A.getDatoRaiz();

  Integer minI = calcularMinimo(A.getHijoIzquierdo(),
      numeroNivel - 1);
  Integer minD = calcularMinimo(A.getHijoDerecho(),
      numeroNivel - 1);

  /* minI y minD pueden ser null */
  if (minI != null && minD != null)
    return (minI < minD) ? minI : minD;
  else
    return (minI == null) ? minD : minI;
}
```

3.2.3. Implementación Java - Recursivo con `int`

```
/* Devuelve Integer.MAX_VALUE cuando no se encuentra */
int calcularMinimo(ArbolBinario<Integer> A, int
    numeroNivel) {
  if (esVacio(A))
    return Integer.MAX_VALUE;
  if (numeroNivel == 0)
    return A.getDatoRaiz();

  int minI = calcularMinimo(A.getHijoIzquierdo(),
      numeroNivel - 1);
  int minD = calcularMinimo(A.getHijoDerecho(),
      numeroNivel - 1);

  return (minI < minD) ? minI : minD;
}
```

3.2.4. Implementación Java - Iterativo

```java
/* Devuelve Integer.MAX_VALUE cuando no se encuentra */
int calcularMinimo(ArbolBinario<Integer> A, int numeroNivel) {
  if (esVacio(A))
    return Integer.MAX_VALUE;

  ColaGenerica<ArbolBinario<Integer>> cola = new ColaGenerica<ArbolBinario<Integer>>();
  int nivelActual = 0, minActual = Integer.MAX_VALUE;
  boolean salir = false;

  /* Recorrido por niveles en el arbol */
  cola.poner(A);
  cola.poner(null);

  while (!cola.esVacia() && !salir) {
    ArbolBinario<Integer> E = cola.sacar();
    if (E == null) {
      if (nivelActual == numeroNivel) {
        salir = true;
      } else {
        nivelActual++;
      }
      if (!cola.esVacia())
        cola.poner(null);
    } else {
      if (nivelActual == numeroNivel && e.getDatoRaiz() < minActual)
        minActual = e.getDatoRaiz();
      if (tieneHijoIzquierdo(A))
        cola.poner(A.getHijoIzquierdo());
      if (tieneHijoDerecho(A))
        cola.poner(A.getHijoDerecho());
    }
  }

  return minActual;
}
```

3.3. Trayectoria Pesada

Se define el valor de trayectoria pesada de una hoja de un árbol binario como la suma del contenido de todos los nodos desde la raíz hasta la hoja multiplicado por el nivel en el que se encuentra. Implemente un método que, dado un árbol binario, devuelva el valor de la trayectoria pesada de cada una de sus hojas.

Considere que el nivel de la raíz es 1. Para el ejemplo de la figura: trayectoria pesada de la hoja 4 es: $(4 * 3) + (1 * 2) + (7 * 1) = 21$.

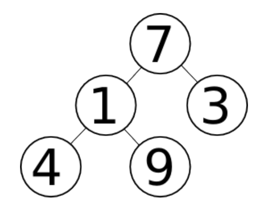


Figura 3.1: Ejercicio 5

3.3.1. Forma de encarar el ejercicio

El ejercicio se puede dividir en tres partes:

1. Recorrido del árbol.
2. Procesamiento de los nodos.
3. Almacenamiento del resultado.

3.3.2. Recorrido del árbol

Existen diferentes maneras de recorrer árboles, cada cual con sus ventajas y desventajas. Un recorrido recursivo puede ser interesante, en especial el preorden o inorden. Cualquiera de éstos se puede aplicar al ejercicio. Por ejemplo, a continuación1 se puede ver el recorrido preorden.

3.3.3. Procesamiento de los nodos

El el dibujo de ejemplo se pueden ver tres hojas. El valor de la trayectoria de cada hoja es:

- Hoja 4: $7 \times 1 + 1 \times 2 + 4 \times 3$.

Algorithm 1 versión en pseudocódigo de un recorrido preorden.

```
function ℝ(A)
    if A no es vacío then
        Procesar(A.dato)
        if A tiene hijo izquierdo then
            ℝ(A.Izq)
        end if
        if A tiene hijo derecho then
            ℝ(A.Der)
        end if
    end if
end function
```

- Hoja 9: $7 \times 1 + 1 \times 2 + 9 \times 3$.

- Hoja 3: $7 \times 1 + 3 \times 2$.

En todas las hojas, como parte del valor de la trayectoria se encuentra la suma parcial de 7×1, y en todos los nodos hijos del nodo 1 se encuentra la suma parcial de 1×2. En general, para un nodo cualquiera del árbol todos sus hijos van a tener como parte de su trayectoria la suma parcial del camino que existe desde la raíz hasta ese nodo.

Esto significa que, recursivamente, podemos acumular el valor del nodo actual multiplicado por el número de nivel y pasarlo por parámetro en el llamado recursivo de ambos hijos.

Actualizando el pseudocódigo2 tenemos el recorrido preorden $\mathbb{R}$ con tres parámetros: el árbol (A), el número de nivel actual (N), y la suma acumulada desde la raíz hasta el nodo actual (V):

3.3.4. Almacenamiento del resultado

En ambas versiones de pseudocódigo anteriores se encuentra el llamado a la función "`Procesar`". El enunciado pide devolver el valor de cada una de las hojas del árbol, por lo que hace falta utilizar una estructura que permite almacenar y devolver múltiples elementos. En nuestra materia, la estructura más adecuada que tenemos se llama `ListaGenerica<T>`, que nos permite almacenar objetos de cualquier tipo que corresponda con el parámetro de comodín de tipo. Otras opciones pueden ser `ListaDeEnteros` o `ColaDeEnteros`.

Hace falta hacer un pequeño análisis acerca de qué debe hacerse en "`Procesar`". El ejercicio pide almacenar el valor de la trayectoria, donde

Algorithm 2 versión en pseudocódigo del procesamiento de nodos.

function $\mathbb{R}(A, N, V)$
  if A no es vacío **then**
    Procesar(A.dato, N, V)
    if A tiene hijo izquierdo **then**
      $\mathbb{R}$(A.Izq, $N + 1$, $V + N \times A$.dato)
    end if
    if A tiene hijo derecho **then**
      $\mathbb{R}$(A.Der, $N + 1$, $V + N \times A$.dato)
    end if
  end if
end function

éste es un valor que se calcula a partir del dato propio de cada nodo y del nivel, teniendo en cuenta que el dato de la hoja también debe ser almacenado. Esto significa dos cosas, por un lado, al hacer el recorrido recursivo el dato no puede almacenarse hasta no visitar una hoja, y por el otro el valor de la hoja debe ser parte del resultado final.

En Java, "`Procesar`" puede ser una llamada a un método separado, o puede ser simplemente el código mismo. Por ejemplo:

```
private void Procesar(ArbolBinario<T> A, int N, int V,
    ListaGenerica<Integer> lista) {
  if (A.esHoja()) {

    lista.agregar(V + (Integer)A.getDatoRaiz() * N);
  }
}
```

3.3.5. Detalles de implementación

Modularización

Modularizar y separar el código en diferentes funciones o métodos suele ser una buena idea. Hasta ahora vimos que el ejercicio se puede separar en dos métodos. Sin embargo, se pide devolver los valores de todas las hojas, por lo que hace falta que el método principal del ejercicio tenga como tipo de retorno `ListaGenerica<T>` (o la estructura elegida). Esto puede complicar la implementación del recorrido preorden, por lo que puede ser de mucha ayuda tener otro método aparte que se encargue del recorrido recursivo. Una buena forma de modularizar este ejercicio puede ser como el siguiente ejemplo:

```java
class EjercicioCinco {
  public static ListaGenerica<Integer> CalcularTrayectoria(ArbolBinario<T> A) {
    ListaGenerica<Integer> lista = new ListaEnlazadaGenerica<Integer>();

    Recorrido(A, 1, 0, lista);

    return lista;
  }

  private static void Recorrido(ArbolBinario<T> A, int N, int V, ListaGenerica<Integer> lista) {
    // Metodo del recorrido "R" preorden recursivo.
    if (!A.esVacio()) {
      // ...
    }
  }

  private static void Procesar(ArbolBinario<T> A, int N, int V, ListaGenerica<Integer> lista) {
    // ...
  }
}
```

`esHoja()`

La definición del método `esHoja()` de la clase `ArbolBinario<T>` puede ser:

```java
class ArbolBinario<T> {
  public boolean esHoja() {
    return !tieneHijoIzquierdo() && !
        tieneHijoDerecho();
  }

  public boolean tieneHijoIzquierdo() {
    return getRaiz().getHijoIzquierdo() != null;
  }

  public boolean tieneHijoDerecho() {
    return getRaiz().getHijoDerecho() != null;
  }
}
```

`public` y `private`

En Java, la forma que se tiene de abstraer y encapsular código y datos es mediante los **especificadores de acceso**. Dos de ellos son `public` y `private`, que permiten definir desde qué parte del código se puede invocar un método o acceder a una variable.

En el caso de este ejercicio, es importante que el método principal del recorrido sea `public` a fin de que los usuarios del `ArbolBinario` lo puedan utilizar. Por otro lado, tener los métodos auxiliares como `private` permite que los detalles de implementación permanezcan ocultos, dejando la libertad que en un futuro sea posible modificar el código y los detalles de implementación teniendo el menor impacto en el mantenimiento del resto del código donde se utilizan estas llamadas.

Clases abstractas

El operador "`new`" solo puede ser usado para instanciar clases que no sean `abstract`. Por ejemplo, el siguiente programa no es correcto, ya que el operador `new` se está usando para instanciar una clase `abstract`.

```java
public abstract class X {
  public X() {
    // ...
  }

  public static void main(String[] args) {
```

```
        X y = new X(); // Error!
    }
}
```

En particular, no es posible crear una instancia de ListaGenerica<T>, ya que esta es una clase abstract. Es necesario instanciar una subclase que no sea abstract. Por ejemplo, ListaEnlazadaGenerica<T>.

```
public void f() {
    ListaGenerica<Integer> L = new
        ListaEnlazadaGenerica<Integer>();
}
```

3.3.6. Errores comunes en las entregas

1. No devolver los valores. El enunciado pide devolver los valores, por lo que imprimirlos mediante System.out.print(); no sería correcto. Hace falta utilizar una estructura que permita almacenar una cantidad variable de elementos para luego devolver la estructura completa.

2. No sumar el valor de la hoja como parte de la trayectoria. Al momento de agregar el valor de la trayectoria hace falta incluir el valor de la hoja multiplicado por su nivel. Manteniendo la idea explicada hasta ahora, el siguiente resultado es incorrecto:

```
private static void Procesar(ArbolBinario<T> A, int
    N, int V, ListaGenerica<Integer> lista) {
  if (A.esHoja()) {
    lista.agregar(V); // Incorrecto, falta calcular
        hoja
  }
}
```

3. No preguntar inicialmente si el árbol es vacío. No solo hace falta preguntar si el arbol tiene hijo izquierdo o hijo derecho, sino también si el arbol original inicial tiene algún dato. Eso se logra preguntando si la raíz de ArbolBinario es null, o mediante el uso del método esVacio().

```
private static void R(ArbolBinario<T> A) {
  if (!A.esVacio()) {
    // ...
  }
}
```

Donde el método `esVacio()` se define dentro de la clase `ArbolBinario` como:

```
class ArbolBinario<T> {
  public boolean esVacio() {
    return getRaiz() == null;
  }
}
```

4. Uso del operador "`++`" a derecha para incrementar de nivel. El operador "`++`" a derecha, por ejemplo en el caso de "`n++`", hace que el valor de la variable `n` se modifique, pero este incremento se realiza después de devolver el valor original. Si se usa `n++` como incremento de nivel, y esta expresión está directamente en el llamado recursivo, el efecto deseado no se cumplirá. En cada llamada recursiva, el parámetro `n` siempre tomará el mismo valor, y no se estará incrementando el nivel.

```
public static void R(ArbolBinario<T> A, int N) {
  if (A.esHoja()) {
    // El parámetro N nunca llega al llamado
       recursivo con el valor incrementado.
  } else {
    if (A.tieneHijoIzquierdo()) R(A.
       getHijoIzquierdo(), N++);
    if (A.tieneHijoDerecho()) R(A.getHijoDerecho(),
       N++);
  }
}
```

5. Pasar por parámetro el árbol en un método no `static` de la clase `ArbolBinario` genera que se esten recorriendo dos árboles a la vez. Por un lado, el árbol que llega por parámetro, y por el otro el arbol de "`this`".

```
class ArbolBinario<T> {
  public ListaGenerica<Integer> Trayectoria(
      ArbolBinario<T> A) {
    // Acá hay dos árboles. "this" y "A".
  }
}
```

La solución a esto puede ser:

a) Indicar los métodos de recorrido como `static`,

b) Quitar el parámetro y hacer el recorrido del arbol con "`this`".

c) Implementar los métodos en una clase diferente a `ArbolBinario`, por ejemplo dentro de la clase "`EjercicioCinco`".

6. Preguntar por la existencia del hijo izquierdo o derecho a partir del valor devuelto por `getHijoIzquierdo()` de `ArbolBinario`.

```
class ArbolBinario<T> {
  public ArbolBinario<T> getHijoIzquierdo() {
    return new ArbolBinario<T>(getRaiz().
        getHijoIzquierdo());
  }
}
```

De acuerdo a la definición vista en clase de `ArbolBinario`, los métodos `getHijoIzquierdo` y `getHijoDerecho` de `ArbolBinario` nunca devuelven `null`, y por lo tanto la siguiente pregunta nunca va a ser `True`, porque el operador "`new`" nunca devuelve `null`.

```
if (arbol.getHijoIzquierdo()==null) {
  // Nunca puede ser True.
}
```

Lo ideal en este caso es utilizar los métodos `tieneHijoIzquierdo` y `tieneHijoDerecho`, implementados dentro de la clase `ArbolBinario` tal como se muestra en la sección *Detalles de implementación*.

7. Variables en `null`. El siguiente código dispara siempre un error de `NullPointerException`. Si la condición del **if** da `True` entonces "`lista`" es `null`, y de ninguna forma es posible utilizar la variable "`lista`" porque justamente su valor es `null`.

```
if(lista==null)
{
   lista.agregar(sum);
}
```

8. Preguntar si `this` es `null`. El lenguaje Java garantiza que "`this`" jamás pueda ser `null`, ya que al momento de invocar un método de instancia en una variable con referencia `null`, se dispara un error de `NullPointerException`.

3.4. Arboles Generales: Ancestro Común más Cercano

Sea un árbol general que representa un árbol genealógico de una familia, donde la raíz es el pariente inmigrante que llegó en barco desde un país lejano, y la relación de hijos son los hijos de cada persona.

Escribir en Java un método que, dado un árbol general de `String`s y dos nombres de personas, devuelva el nombre del ancestro en común más cercano a ambas personas. Esto es, la raíz siempre será ancestro en común de cualquier par de nodos del árbol, pero no necesariamente es el más cercano.

Asuma que no hay dos personas que se llamen igual en la familia (tradición familiar), y en caso que no exista relación en común entre ambas personas, por ejemplo que una de las personas no figure en el árbol, devolver `null`.

```
String ancestroComun(ArbolGeneral<String> A, String
    Nombre1, String Nombre2) { ... }
```

3.4.1. Explicación General

El problema se reduce a encontrar el nodo con mayor número de nivel que los elementos indicados formen parte de los nodos de sus subárboles, sin importar en qué parte del subárbol se encuentren.

Si alguno de los dos elementos indicados es la raíz misma, dado que la raíz no tiene ancestro deberá devolverse `null`. Además, pueden existir varios nodos que posean ambos elementos en sus subárboles, por ejemplo la raíz es ancestro común de cualquier par de nodos del árbol, es por eso que se busca el de mayor nivel.

3.4.2. Versión Recursiva

```
class X {
  public static String ancestroComun(ArbolGeneral<
      String> A, String Nom1, String Nom2) {
    if (A.esVacio())
      return null;

    boolean tieneN1 = false, tieneN2 = false;
    ListaGenerica<ArbolGeneral<String>> L = A.getHijos
        ();
    String ancestro = null;
```

```java

    // Primero me fijo si algun subárbol tiene un nodo ancestro.
    L.comenzar();
    while (!L.fin() && ancestro == null) {
      ancestro = ancestroComun(L.elemento(), Nom1, Nom2);
      L.proximo();
    }

    // En caso de tenerlo, devuelvo este, ya que hay que devolver el de mayor nivel.
    if (ancestro != null)
      return ancestro;

    // Si no hay hijo que sea ancestro me fijo si el nodo actual es ancestro; donde N1 y N2 ambos esten en alguno de mis subárboles.
    L.comenzar();
    while (!L.fin()) {
      tieneN1 = tieneN1 || esHijo(L.elemento(), Nom1);
      tieneN2 = tieneN2 || esHijo(L.elemento(), Nom2);
      if (tieneN1 && tieneN2)
        break;
      L.proximo();
    }

    if (tieneN1 && tieneN2)
      return A.getDatoRaiz();
    return null;
  }

  private static boolean esHijo(ArbolGeneral<String> A, String dato) {
    // Devuelve True si algun nodo del subarbol total posee como dato el parametro "Dato"
    if (A.esVacio())
      return false;
    if (A.getDatoRaiz().equals(dato))
      return true;
```

```
      ListaGenerica<ArbolGeneral<String>> L = A.getHijos
          ();
      L.comenzar();
      while (!L.fin()) {
         if (esHijo(L.elemento(), dato)) {
            return true;
         }
         L.proximo();
      }
      return false;
   }
}
```

Capítulo 4

Árboles AVL

4.1. Perros y Perros y Gatos

A partir de una lista de palabras, implemente en Java una función que, de forma *eficiente* y usando un AVL, determine la cantidad de palabras diferentes en la secuencia.

Ejemplo: si la secuencia es: [”Perro”, ”Gato”, ”Perro”, ”Perro”, ”Gato’], la función deberá devolver 2 porque si bien hay 5 elementos, solo ”Perro” y ”Gato” son las dos palabras diferentes que aparecen.

4.1.1. Explicación del problema

Hay que descartar las palabras repetidas y contar las diferentes. Eso se logra recorriendo la lista de entrada; por cada palabra agregarla al árbol solo si no se encuentra, contando la cantidad de palabras agregadas o devolviendo el tamaño final del árbol.

4.1.2. Implementación Java

```
int contar(ListaGenerica<String> L) {
  ArbolAVL<String> A = new ArbolAVL<String>();
  int cantidad = 0;

  L.comenzar();
  while (!L.fin()) {
    if (!A.incluye(L.elemento())) {
      A.insertar(L.elemento());
      cantidad++;
    }
    L.proximo();
  }

  return cantidad;
}
```

4.2. Pepe y Alicia

Pepe y Alicia quieren saber cuáles son sus intereses en común. Para ello, cada uno armó una lista de intereses, pero llegaron a tener miles cada uno.

Implemente en Java un programa que dadas dos listas de `String` devuelva de forma *eficiente* una lista con los intereses en común, sin elementos repetidos. Los intereses de cada uno pueden estar repetidos, pero el resultado debe tener uno solo de cada tipo. No hace falta implementar las estructuras.

Ejemplo: A Pepe le gusta $[A, B, C, A, B]$ y a Alicia $[A, D, A]$. Sus intereses en común son: $[A]$.

```
ListaGenerica<String> interesesComun(ListaGenerica<
    String> P, ListaGenerica<String> A) { ... }
```

4.2.1. Explicación del problema

Es una intersección entre dos conjuntos, devolviendo los elementos que pertenecen a ambas listas. Hace falta tener en cuenta que el enunciado pide que sea de forma *eficiente*. Es por ello que hace falta utilizar alguna estructura que permita mejorar el tiempo de ejecución.

Una primer solución puede ser recorrer la primer lista, y por cada elemento recorrer la segunda lista verificando si se encuentra en esta. Por ejemplo:

```
ListaGenerica<String> interesesEnComun(ListaGenerica<
    String> P, ListaGenerica<String> A) {
  /* Solucion ineficiente */
  ListaGenerica<String> resultado = new ListaGenerica<
      String>();
  for(P.comenzar(); !P.fin(); P.siguiente()) {
    String E = P.elemento();
    if (A.incluye(E) && !resultado.incluye(E))
      resultado.agregar(P.elemento());
  }
  return resultado;
}
```

Sin embargo, esta solución no es eficiente ya que por cada elemento de la primer lista se recorre la segunda lista, dando un orden de ejecución $O(n^2)$, ya que la función `incluye()` de la clase `ListaGenerica<T>` debe recorrer toda la estructura preguntando por cada elemento.

Para mejorar el tiempo de ejecución, una mejor solución puede ser utilizar una estructura de datos intermedia, por ejemplo un árbol AVL.

Hace falta utilizar dos árboles AVL. El primer árbol contendrá los elementos de la primer lista, permitiendo conocer de forma eficiente qué elemento se encuentran en ésta lista. Primero, hace falta guardar los elementos de la primer lista en el primer árbol AVL sin repetición. Luego, recorriendo la segunda lista se pregunta si cada elemento de esta se encuentra en el primer arbol pero no en el segundo. En caso de que se cumpla la condición, se inserta en el segundo árbol AVL y además en una lista temporal.

La lista temporal contendrá los elementos que se encuentran en ambas listas iniciales, sin repetición, y será construida en tiempo $O(n \log n)$, con n siendo la suma de la cantidad de elementos de ambas listas iniciales. El segundo árbol contendrá los elementos que se encuentran en la lista temporal, permitiendo una búsqueda eficiente para verificar si ya fueron agregados a la lista resultante.

4.2.2. Implementación Java

```
class Intereses {
  ListaGenerica<String> interesesComun(ListaGenerica<String> P, ListaGenerica<String> A) {
    ListaGenerica<String> resultado = new ListaEnlazadaGenerica<String>();
    ArbolAVL<String> lasDePepe = new ArbolAVL<String>();
    ArbolAVL<String> enComun = new ArbolAVL<String>();

    P.comenzar();
    while (!P.fin()) {
      if (!lasDePepe.incluye(P.elemento())) {
        lasDePepe.agregar(P.elemento());
      }
      P.siguiente();
    }

    A.comenzar();
    while (!A.fin()) {
      if (lasDePepe.incluye(A.elemento()) && !enComun.incluye(A.elemento())) {
        enComun.agregar(A.elemento());
        resultado.agregar(A.elemento());
      }
```

```
            A.siguiente();
        }

        return resultado;
    }
}
```

Capítulo 5

Grafos

5.1. Recorridos sobre grafos

Existen dos tipos de recorridos sobre grafos: recorrido en profundidad (DFS), y recorrido en amplitud (BFS).

El recorrido tipo DFS se ejecuta de forma recursiva sobre cada nodo del grafo haciendo una búsqueda exhaustiva por cada posible camino que se puede tomar entre un nodo inicial y el resto de los nodos del grafo.

La forma general de un recorrido DFS es:

```
void DFS(Vertice V, int peso) {
  visitados[V.posicion()] = true;

  ListaGenerica ADY = V.obtenerAdyacentes();
  for (ADY.comenzar(); !ADY.fin(); ADY.proximo()) {
    Arista E = ADY.elemento();
    if (!visitados[E.verticeDestino().posicion()])
      DFS(E.verticeDestino(), peso + E.peso());
  }

  visitados[V.posicion()] = false;
}
```

Es importante tener en cuenta que para poder hacer la búsqueda de forma exhaustiva, por cada adyacente de cada nodo hace falta iniciar de forma recursiva el recorrido. De esta forma, se evalúan todas las posibles combinaciones de nodos y aristas posibles del grafo, partiendo desde un vértice inicial.

También es importante marcar los nodos como visitados al iniciar la recursión, preguntar si el nodo no se encuentra actualmente visitado en el recorrido de adyacentes, y por último desmarcar el visitado al terminar la recursión.

A diferencia del DFS, en un recorrido BFS no se evalúan todas las posibles combinaciones de caminos. En un DFS, cada vértice puede ser visitado más de una vez, mientras que en un BFS cada vértice puede ser visitado a lo sumo una única vez.

Por un lado, esto permite que en general los recorridos BFS sean más eficientes que un recorrido DFS. Por otro lado, no siempre es posible resolver cualquier recorrido con un BFS, no dejando otra opción que hacerlo aplicando un recorrido DFS.

La forma general de un recorrido BFS es:

```
void BFS(Vertice V) {
```

```
    ColaGenerica<Vertice> C = new ColaGenerica<Vertice>();
    C.poner(V);

    while (!C.esVacia()) {
        Vertice W = C.sacar();
        ListaGenerica ADY = W.obtenerAdyacentes();
        for (ADY.comenzar(); !ADY.fin(); ADY.proximo()) {
            Arista E = ADY.elemento();
            if (!visitados[E.verticeDestino().posicion()]) {
                visitados[E.verticeDestino().posicion()] = true;
                C.poner(E.verticeDestino());
            }
        }
    }
}
```

Para calcular el camino mínimo en un grafo no pesado es posible utilizar un recorrido `BFS`, modificando el anterior ejemplo general de forma que además de marcar como true el adyacente en visitado, también actualizar la distancia. Hacer un recorrido `BFS` en un grafo no pesado **garantiza** que al finalizar el recorrido la tabla de distancias tendrá el camino mínimo siempre.

También es posible realizar un recorrido `DFS` para hacer el cálculo del camino mínimo en un grafo no pesado, teniendo en cuenta la desventaja de que el tiempo de ejecución será mayor con respecto a un recorrido `BFS`.

Sin embargo, en un grafo pesado, ya sea con aristas positivas o negativas, en general no es posible calcular el camino mínimo mediante el algoritmo general de `BFS`.

En un grafo pesado con aristas todas no negativas, hace falta utilizar el Algoritmo de Camino Mínimo de Dijkstra, reemplazando la `ColaGenerica` por una cola de prioridad, y teniendo el peso del vértice calculado hasta el momento como valor de prioridad del nodo. Utilizar una `Heap` como estructura de cola de prioridad permite implementar el algoritmo de Dijkstra de la forma más eficiente, $O(n \log n)$.

De otro modo, si el grafo es pesado con aristas negativas, es necesario utilizar un recorrido `DFS` que verifique exhaustivamente todas las posibles combinaciones de caminos del grafo.

Por ejemplo, para calcular el camino mínimo en un grafo pesado mediante un `DFS`:

```
void DFS(Vertice V, int peso) {
```

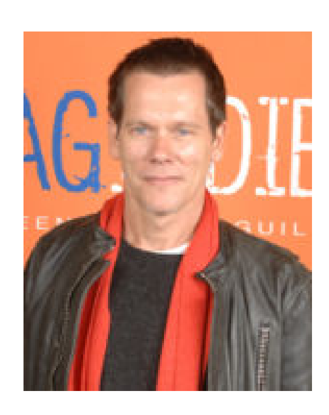
```
    visitados[V.posicion()] = true;

    if (distancias[V.posicion()] > peso) {
        distancias[V.posicion()] = peso;
    }

    ListaGenerica ADY = V.obtenerAdyacentes();
    for (ADY.comenzar(); !ADY.fin(); ADY.proximo()) {
        Arista E = ADY.elemento();
        if (!visitados[E.verticeDestino().posicion()])
            DFS(E.verticeDestino(), peso + E.peso());
    }

    visitados[V.posicion()] = false;
}
```

5.2. El número Bacon



> Kevin Norwood Bacon (n. el 8 de julio de 1958) es un actor de cine y teatro estadounidense notable por sus papeles en National Lampoon's Animal House, Diner, The Woodsman, Friday the 13th, Hollow Man, Tremors y Frost/Nixon.
>
> Bacon ha ganado un Premios Globo de Oro y Premios del Sindicato de Actores, estuvo nominado para un Premio Emmy, y fue nombrado por The Guardian, como uno de los mejores actores que no ha recibido una nominación al Premio Óscar.[1]

Hace algunos años en una entrevista, Kevin Bacon comentó que él había actuado con la mayoría de los actores de Hollywod, ya sea en la misma película o indirectamente con otro actor.

Ahí nació el "Número de Bacon", con la idea de establecer cual era la distancia entre Bacon y el resto de los actores de Hollywood[2]. Los actores que habían actuado en alguna película junto a Bacon tienen el número 1, los que actuaron en alguna película junto a alguien que actuó con Bacon tienen el número 2, y así sucesivamente.

Por ejemplo, el actor Sean Penn tiene como Número de Bacon el 1 ya que actuaron juntos en la película Río Místico, mientras que Sean Connery tiene como Número de Bacon el 2, ya que si bien no actuaron juntos en ninguna película, ambos actuaron junto al actor John Lithgow en al menos película (Sean Connery actuó en "A Good Man in Africa", y Kevin Bacon en "Footloose". John Lithgow actuó en ambas películas.)

[1] Imagen y texto de Wikipedia, http://es.wikipedia.org/wiki/Kevin_Bacon
[2] Bacon tomó esta idea y fundó una organización de caridad, www.sixdegrees.org

5.2.1. Calcular el Número de Bacon

The Oracle Of Bacon[3] es una página web que permite ver cuál es la relación entre dos actores a partir de las películas que ambos actuaron. ¿Cómo funciona?

5.2.2. Explicación del problema

Dado el nombre de un actor, indicar cual es su Número de Bacon. Eso es, cuál es la mínima distancia que se puede establecer entre Kevin Bacon y el segundo actor, relacionados por las películas en las que actuaron.

El problema se puede modelar como un grafo, teniendo a cada actor de Hollywood como un nodo del grafo, y cada película en la que actuaron juntos como una arista del grafo.

Entonces el Número de Bacon se puede calcular con un recorrido sobre el grafo, calculando la mínima distancia entre los dos actores. Eso se logra facilmente con un algoritmo de camino mínimo en un grafo no pesado, ya sea BFS o DFS.

5.2.3. Implementación Java - BFS

```java
class BFS {
  /* Buscar el vertice del actor */
  private Vertice<String> buscarActor(Grafo<String> G,
      String actor) {
    ListaGenerica<Vertice<String>> vertices = G.
        listaDeVertices();
    vertices.comenzar();
    while (!vertices.fin()) {
      Vertice<String> v = vertices.elemento().
          verticeDestino();
      if (v.dato().equals(actor))
        return v;
      vertices.proximo();
    }
    return null;
  }

  /* Camino minimo en un grafo no pesado con BFS */
```

[3] http://oracleofbacon.org

```
  int numeroDeBacon(Grafo<String> G, String actor) {
    Vertice<String> inicial = buscarActor(G, "Kevin Bacon");
    Vertice<String> destino = buscarActor(G, actor);

    int N = G.listaDeVertices().tamanio();
    ColaGenerica<Vertice<String>> c = new ColaGenerica<Vertice<String>>();
    boolean visitados[] = new boolean[N];
    int distancias[] = new int[N];

    for (int i = 0; i < N; ++i) {
      visitados[i] = false;
      distancias[i] = Integer.MAX_VALUE;
    }

    distancias[inicial.posicion()] = 0;
    cola.poner(inicial);

    while (!cola.esVacia()) {
      Vertice<String> v = cola.sacar();

      if (v == destino) /* Actor encontrado */
        return distancias[v.posicion()];

      ListaGenerica<Arista<String>> adyacentes = v.obtenerAdyacentes();
      adyacentes.comenzar();
      while (!adyacentes.fin()) {
        Vertice<String> w = adyacentes.elemento().verticeDestino();
        int nuevaDist = distancias[v.posicion()] + 1;

        if (!visitado[w.posicion()]) {
          visitado[w.posicion()] = true;
          distancias[w.posicion()] = nuevaDist;
          cola.poner(w);
        }
        adyacentes.proximo();
      }
    }
```

```java

    /* El actor no esta relacionado por ninguna pelicula con Kevin Bacon */
    return Integer.MAX_VALUE;
  }
}
```

5.2.4. Implementación Java - DFS

```java
class DFS {
  private boolean visitados[];
  private int distancias[];

  /* Buscar el vertice del actor */
  private Vertice<String> buscarActor(Grafo<String> G, String actor) {
    ListaGenerica<Vertice<String>> vertices = G.listaDeVertices();
    vertices.comenzar();
    while (!vertices.fin()) {
      Vertice<String> v = vertices.elemento().verticeDestino();
      if (v.dato().equals(actor))
        return v;
      vertices.proximo();
    }
    return null;
  }

  /* Camino minimo en un grafo no pesado con DFS */
  public int numeroDeBacon(Grafo<String> G, String actor) {
    Vertice<String> inicial = buscarActor(G, "Kevin Bacon");
    Vertice<String> destino = buscarActor(G, actor);

    int N = G.listaDeVertices().tamanio();
    ColaGenerica<Vertice<String>> c = new ColaGenerica<Vertice<String>>();
    visitados[] = new boolean[N];
    distancias[] = new int[N];
```

```java

        for (int i = 0; i < N; ++i) {
            visitados[i] = false;
            distancias[i] = Integer.MAX_VALUE;
        }

        numeroDeBaconDFS(inicio, 0);

        return distancias[destino.posicion()];
    }

    private void numeroDeBaconDFS(Vertice<String> v, int peso) {
        visitados[v.posicion()] = true;

        if (peso < distancias[v.posicion()]) {
            distancias[v.posicion()] = peso;
        }

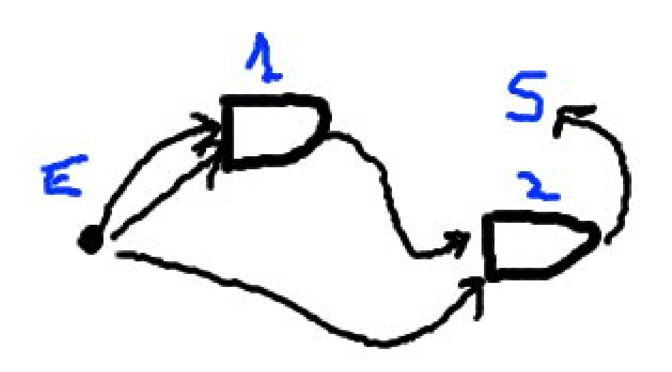
        ListaGenerica<Arista<String>> adyacentes = v.obtenerAdyacentes();
        adyacentes.comenzar();
        while (!adyacentes.fin()) {
            Vertice<String> w = adyacentes.elemento().verticeDestino();

            if (!visitado[w.posicion()])
                numeroDeBaconDFS(w.verticeDestino(), peso + 1);

            adyacentes.proximo();
        }

        visitados[v.posicion()] = false;
    }
}
```

5.3. Juan Carlos Verticio



Juan Carlos Verticio es un artista que vive en la ciudad de Grafonia, y entre sus pasiones personales está la de hacer dibujos de grafos[4], y recorrer la ciudad en bicicleta. Verticio desea entrenar para la final de ciclistas 2012, patrocinada por la División de Finanzas y Seguridad (`DFS`).

Entre sus planes de entrenamiento, Verticio desea armar una serie de recorridos por la ciudad. A fin de optimizar su tiempo en sus ratos libres en la semana, él quiere poder salir y hacer un recorrido entre dos esquinas de la ciudad.

Para ello necesita conocer cuáles caminos tienen *exactamente* una determinada longitud entre dos esquinas. El camino no puede terminar a mitad de cuadra. Por ejemplo, todos los caminos de la ciudad que entre dos esquinas miden 21 km.

A partir de un grafo dirigido[5] que representa la ciudad de Grafonia, realizar una función en Java que le permita a Verticio obtener una lista de todos los caminos de la ciudad con una longitud específica. Las aristas del grafo tienen todas valores positivos, siendo la distancia de las diferentes calles de la ciudad, medidas en Km como números enteros. Los vértices del grafo contienen un `String` que representa la esquina de la ciudad.

5.3.1. Explicación del problema

Hace falta recorrer el grafo de forma de obtener todos los caminos posibles con una determinada longitud. La forma más directa de resolverlo es un con un `DFS`, probando todas las posibles combinaciones de aristas del grafo.

[4] La imagen es una de sus más recientes obras.
[5] Verticio no puede ir por las calles en contramano.

Para ello, por cada nodo del grafo hace falta iniciar un DFS, llevando la lista de esquinas de la ciudad y el costo actual acumulado.

5.3.2. Implementación Java - DFS

```
class DFS {
  boolean visitados[];
  int distancias[];

  public ListaGenerica<ListaGenerica<String>> buscarCaminosDeCosto(Grafo<String> G, int costo) {
    int N = G.listaDeVertices().tamanio();
    visitados[] = new boolean[N];
    distancias[] = new int[N];
    ListaGenerica<ListaGenerica<String>> resultado = new ListaGenerica<ListaGenerica<String>>();

    for (int i = 0; i < N; ++i) {
      visitados[i] = false;
      distancias[i] = Integer.MAX_VALUE;
    }

    for (int i = 0; i < N; ++i) {
      ListaGenerica<String> caminoActual = new ListaGenerica<String>();
      Vertice<String> v = G.listaDeVertices().elemento(i);
      buscarCaminosDeCostoDFS(v, resultado, caminoActual, costo, 0);
    }

    return resultado;
  }

  private void buscarCaminosDeCostoDFS(Vertice<String> v, ListaGenerica<ListaGenerica<String>> resultado, ListaGenerica<String> caminoActual, int costo, int costoActual) {
    visitados[v.posicion()] = true;
    caminoActual.agregar(v.dato(), caminoActual.tamanio
```

```
        ());

    if (costo == costoActual) {
      /* Crear una copia de la lista */
      ListaGenerica<String> nuevaLista = new
          ListaGenerica<String>();
      caminoActual.comenzar();
      while (!caminoActual.fin()) {
        nuevaLista.agregar(caminoActual.elemento(),
            nuevaLista.tamanio());
        caminoActual.proximo();
      }
      resultado.agregar(nuevaLista);
    }

    ListaGenerica<Arista<String>> adyacentes = v.
        obtenerAdyacentes();
    adyacentes.comenzar();
    while (!adyacentes.fin()) {
      Arista<String> a = adyacentes.elemento();
      Vertice<String> w = a.verticeDestino();

      if (!visitados[w.posicion()]) {
        buscarCaminosDeCostoDFS(w, resultado,
            caminoActual, costo, costoActual + a.peso())
            ;
      }

      adyacentes.proximo();
    }

    visitados[v.posicion()] = false;
    caminoActual.eliminar(caminoActual.tamanio() - 1);
  }
}
```

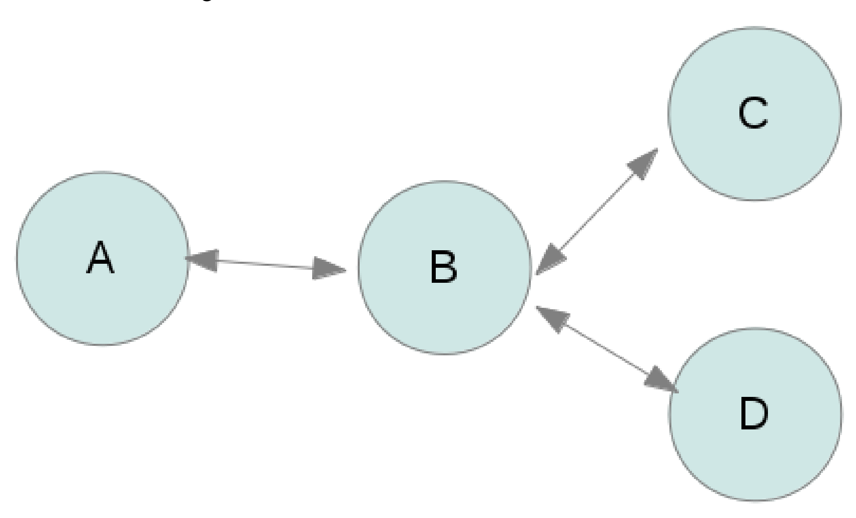
5.4. Virus de Computadora

Un poderoso e inteligente virus de computadora infecta cualquier computadora en 1 minuto, logrando infectar toda la red de una empresa con cientos de computadoras. Dado un grafo que representa las conexiones entre las computadoras de la empresa, y una computadora ya infectada, escriba un programa en Java que permita determinar el tiempo que demora el virus en infectar el resto de las computadoras.

Asuma que todas las computadoras pueden ser infectadas, no todas las computadoras tienen conexión directa entre sí, y un mismo virus puede infectar un grupo de computadoras al mismo tiempo sin importar la cantidad.

Por ejemplo, si la computadora A se conecta con la B, y la B solo con la C y D, el tiempo total desde la A será de dos minutos: la A ya está infectada, un minuto para la B, y un minuto para la C y D (ambas C y D al mismo tiempo).

5.4.1. Dibujo



5.4.2. Características del grafo

Es un grafo no dirigido con aristas sin peso.

5.4.3. Explicación del problema

El problema empieza con una primer computadora infectada, y hay que simular la infección en el resto de la red. Cada vez que el virus "salta"desde una computadora hacia la siguiente transcurre un minuto, por lo que si hace

cuatro saltos transcurren cuatro minutos. En este caso, el tiempo total de infección de la red es de cuatro minutos.

Hay que tener en cuenta que una misma computadora se puede infectar por diferentes caminos, y dado que el virus infecta varias computadoras al mismo tiempo, la infección termina siendo por la que está más cerca.

El problema se reduce a un camino mínimo en un grafo no pesado, y el tiempo total de infección de la red es el mayor costo de los nodos. La solución más directa es un BFS, que representa la forma en la que el virus se distribuye por la red.

5.4.4. Implementación Java - BFS

```java
class BFS {
  public int calcularTiempoInfeccion(Grafo G, Vertice
      inicial) {
    int N = G.listaDeVertices().tamanio();
    Cola c = new Cola();
    boolean visitados[] = new boolean[N];
    int distancias[] = new int[N];
    int maxDist = 0;

    for (int i = 0; i < N; ++i) {
      visitados[i] = false;
      distancias[i] = Integer.MAX_VALUE;
    }

    distancias[inicial.posicion()] = 0;
    cola.poner(inicial);

    while (!cola.esVacia()) {
      Vertice v = cola.sacar();
      ListaGenerica<Arista> adyacentes;

      adyacentes = v.obtenerAdyacentes();
      adyacentes.comenzar();
      while (!adyacentes.fin()) {
        Vertice w = adyacentes.elemento().
            verticeDestino();
        int nuevaDist = distancias[v.posicion()] + 1;

```

```
            if (!visitado[w.posicion()]) {
               visitado[w.posicion()] = true;
               distancias[w.posicion()] = nuevaDist;
               if (nuevaDist > maxDist)
                  maxDist = nuevaDist;
               cola.poner(w);
            }
            adyacentes.proximo();
         }
      }

      return maxDist;
   }
}
```

5.5. Circuitos Electrónicos

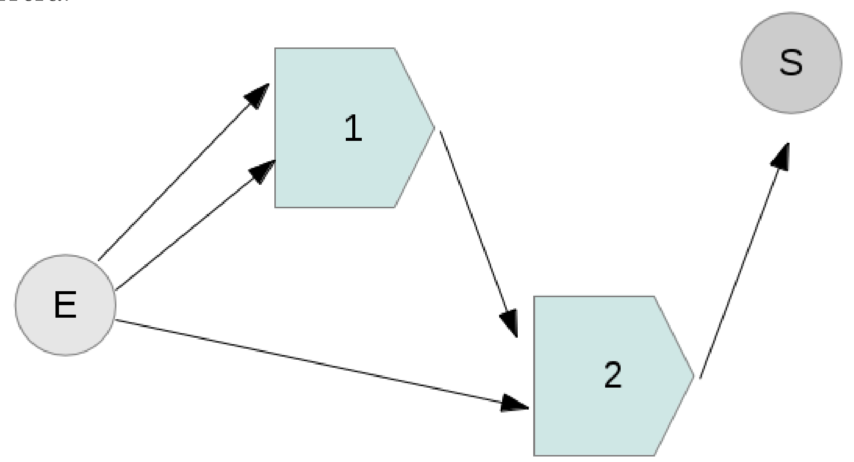
En un circuito electrónico digital la electricidad viaja a la velocidad de la luz. Sin embargo, las diferentes compuertas tienen un tiempo de demora entre que la señal llega por sus entradas y se genera el resultado.

Si el circuito está mal diseñado, cuando se conectan varias compuertas entre sí ocurre lo que se llama una condición de carrera, donde las dos señales que le llegan a una compuerta NAND lo hacen a diferente tiempo, y la salida de la compuerta tardará en estabilizarse. Asuma que tiempo que demoran las señales en viajar por los cables es cero, y el tiempo que demoran las señales en atravezar una compuerta es el mismo para todas.

Se tiene un grafo que representa el circuito electrónico, y una única entrada que generan una señal.

5.5.1. Dibujo

El siguiente es un dibujo de un grafo que contenga una condicion de carrera.



5.5.2. Características del grafo

Es un grafo dirigido no pesado. Hay tres tipos de nodos: el nodo entrada que genera las señales, los nodos de salida que reciben las señales, y los nodos de compuertas, que siempre tienen dos aristas de entrada y una arista de salida.

5.5.3. Explicación del problema

Hay una carrera cuando una señal le llega a una compuerta más rápido por una de las entradas que por las otras.

Por ejemplo, en el dibujo anterior están las compuertas 1 y 2, una entrada por donde se disparan las señales, y una salida, por donde queda el resultado de las compuertas. Las señales que le llegan a la compuerta 1 son todas al mismo momento de tiempo, pero las señales que le llegan a la compuerta 2 lo hacen a diferente tiempo, porque una de las entradas es la salida de la compuerta 1.

La idea del ejercicio es hacer el seguimiento de los diferentes caminos que pueden tomar la señal. No es necesario hacer una lista de todos, sino encontrar el caso de la primer carrea.

Una forma de ver la solución es simular el paso del tiempo en el circuito, haciendo el mismo recorrido que las señales a lo largo del tiempo. Eso se puede hacer de dos formas: BFS o DFS.

En el caso del DFS, por cada nodo que se visita se va incrementando en 1 un contador a medida que se llama a la recursión, este valor se lo almacena en una tabla, y si cuando se vuelve a visitar un nodo por segunda vez el valor es diferente al primero, hay una carrera.

En el BFS es similar, pero hace falta diferenciar correctamente los nodos visitados de los que no lo están. Tambien, es necesario mantener una tabla; si el nodo no fue visitado actualizr el valor como el valor del padre + 1, y si el nodo ya fue visitado verificar que el valor es el mismo que el del padre + 1. Esto se hace dentro del while que recorre adyacentes.

En ambos casos, el contador sirve para ver en qué instante de tiempo le están llegando las señales a la compuerta. Si por un lado le llega un valor diferente al que le llega por el otro, quiere decir que le llegan a diferente tiempo y entonces hay una carrera.

5.5.4. Implementación Java - BFS

El siguiente programa Java determina si un circuito electrónico presenta una codición de carrea.

```
class BFS {
  public boolean hayCarrera(Grafo G, Vertice inicial) {
    int N = G.listaDeVertices().tamanio();
    Cola c = new Cola();
    boolean visitados[] = new boolean[N];
    int distancias[] = new int[N];

```

```
    for (int i = 0; i < N; ++i) {
      visitados[i] = false;
      distancias[i] = Integer.MAX_VALUE;
    }

    distancias[inicial.posicion()] = 0;
    cola.poner(inicial);

    while (!cola.esVacia()) {
      Vertice v = cola.sacar();
      ListaGenerica<Arista> adyacentes;

      adyacentes = v.obtenerAdyacentes();
      adyacentes.comenzar();
      while (!adyacentes.fin()) {
        Vertice w = adyacentes.elemento().
          verticeDestino();
        int nuevaDistancia = distancias[v.posicion()] +
          1;

        if (!visitados[w.posicion()]) {
          visitados[w.posicion()] = true;
          distancias[w.posicion()] = nuevaDistancia;
          cola.poner(w);
        } else {
          if (distancias[w.posicion()] !=
            nuevaDistancia)
            return true; /* Hay carrera */
        }
        adyacentes.proximo();
      }
    }
    return false;
  }
}
```

5.5.5. Implementación Java - DFS

```
class DFS {
  boolean visitados[];
```

```
  int distancias[];

  public boolean hayCarrera(Grafo<Integer> G, Vertice<Integer> inicial) {
    int N = G.listaDeVertices().tamanio();
    visitados[] = new boolean[N];
    distancias[] = new int[N];

    for (int i = 0; i < N; ++i) {
      visitados[i] = false;
      distancias[i] = Integer.MAX_VALUE;
    }

    return hayCarreraDFS(inicial, 0);
  }

  private boolean hayCarreraDFS(Vertice<Integer> v, int tiempo) {
    visitados[v.posicion()] = true;

    if (distancias[v.posicion()] != Integer.MAX_VALUE) {
      if (distancias[v.posicion()] != tiempo) {
        visitados[v.posicion()] = false;
        return true;
      }
    } else {
      distancias[v.posicion()] = tiempo;
    }

    ListaGenerica<Vertice<Integer>> adyacentes = v.obtenerAdyacentes();
    adyacentes.comenzar();
    while (!adyacentes.fin()) {
      Vertice<Integer> w = adyacentes.elemento().verticeDestino();
      if (!visitados[w.posicion()] && hayCarreraDFS(w, tiempo + 1))
        visitados[v.posicion()] = false;
        return true;
      }
```

```
            adyacentes.proximo();
        }

        visitados[v.posicion()] = false;
        return false;
    }
}
```

5.6. Viajante de Encuestas

La facultad de Ciencias Exactas tiene docentes dando clases en diferentes Facultades: Informática, Química, Astronomía, Medicina, etc. María es contratada por Exactas para realizar encuestas a los alumnos de las diferentes comisiones de los diferentes cursos de **todas las facultades**, y quiere saber cuál es la ruta óptima que menos tiempo le lleve para visitar **todos los cursos** de los diferentes edificios.

Dado un grafo con vértices que representan los diferentes cursos de todas las facultades, y con aristas representando cuánto cuesta llegar desde un curso a otro, haga un programa en Java que le ayude a María definir en qué orden visitar los diferentes cursos para que le cueste lo menos posible. Asuma que el primer curso lo puede elegir sin costo, y los cursos se pueden visitar en cualquier momento.

Ayuda: Implemente un recorrido sobre el grafo que pruebe las diferentes variantes de caminos.

5.6.1. Explicación del problema

Este problema se conoce como Viajante de Comercio, TSP en inglés. Es un problema `NPH`, una categoría de problemas[6] de la que no se conoce un algoritmo que funcione eficiente en grafos con cientos de miles (o más) de nodos.

De todas formas, es muy fácil ver que se puede resolver con un DFS, iniciando un recorrido desde cada nodo del grafo y probando todas las posibles combinaciones de caminos.

A medida que se va haciendo la recursión se va creando el camino, y una vez cumplida la condición de visitar todos los nodos se verifica si el camino actual tiene el menor costo entre los ya procesados.

Es importante tener en cuenta que el grafo debe ser conexo. De otra forma nunca se cumplirá la condición de visitar todos los nodos del grafo.

5.6.2. Implementación Java

```java
class ViajanteDeEncuestas {
  int N;
  boolean[] visitados;
  ListaGenerica<String> mejorCamino;
```

[6] Categorizar problemas como `P`, `NP`, `NPC`, `NPH` es un tema que se encuentra fuera del alcance de este curso.

```
int mejorCaminoCosto = Integer.MAX_VALUE;

ListaGenerica<String> calcularMejorViaje(Grafo<String> G) {
  N = G.listaDeVertices().tamanio();
  visitados = new boolean[N];

  for (int i = 0; i < N; ++i)
    visitados[i] = false;

  for (int i = 0; i < N; i++) {
    ListaGenerica<String> caminoActual = new ListaEnlazadaGenerica<String>();
    calcularCaminoMinimoDFS(G.listaDeVertices().elemento(i), caminoActual, 1, 0);
  }
  return mejorCamino;
}

void calcularCaminoMinimoDFS(Vertice<String> V, ListaGenerica<String> caminoActual, int cantVisit, int costoActual) {
  visitados[V.posicion()] = true;
  caminoActual.agregar(V.dato(), caminoActual.tamanio());

  if (cantVisit == N && costoActual < mejorCaminoCosto)
  {
    /* Copiar la lista */
    ListaGenerica<String> nuevaLista = new ListaEnlazadaGenerica<String>();
    caminoActual.comenzar();
    while (!caminoActual.fin()) {
      nuevaLista.agregar(nuevaLista, nuevaLista.tamanio());
      caminoActual.proximo();
    }
    /* Actualizar el camino minimo */
    mejorCamino = nuevaLista;
    mejorCaminoCosto = costoActual;
```

```
      }
      else
      {
         /* Seguir DFS en los adyacentes */
         ListaGenerica<String> adyacentes = V.
             obtenerAdyacentes();
         adyacentes.comenzar();
         while (!adyacentes.fin()) {
            Arista<String> A = adyacentes.elemento();
            Vertice<String> W = A.verticeDestino();
            if (!visitados[W.posicion()]) {
               calcularCaminoMinimoDFS(W, cantVisit + 1,
                   costoActual + A.peso());
            }
            adyacentes.proximo();
         }
      }

      caminoActual.eliminar(caminoActual.tamanio() - 1);
      visitados[V.posicion()] = false;
   }
}
```